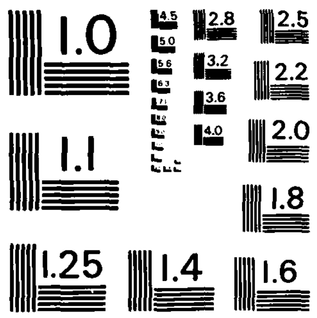
AD-A148 341   A MODULAR INTRODUCTION TO ADA(U) BROWN UNIV PROVIDENCE   1/1
RI   P WEGNER 10 OCT 83 N00014-78-C-0656

UNCLASSIFIED   F/G 9/2   NL

END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1)

AD-A148 341

DTIC FILE COPY

A Modular Introduction to Ada *

Peter Wegner, Brown University

Oct. 1983

This modular introduction to Ada includes seven instruction units each consisting of a sequence of frames:

(1) Lexical Elements (8 frames),
(2) Syntactic Notation (4 Frames),
(3) Expressions, Statements, and Declarations (7 frames),
(4) Introduction to Types (7 frames),
(5) Control Structures (8 frames),
(6) Arrays (6 frames),
(7) Subprograms and Parameter Passing (7 frames).

It provides an introduction to low-level Ada concepts that can serve as a foundation for the study of higher-level concepts such as packages and tasks. It complements top-down introductions to Ada ~~such as that of Booch [1]~~, and can be used for self study prior to taking a course on Ada. Its style is that of the Self-Assessment Procedure ~~[2]~~, but it is more elementary. Its discussion of Ada syntax and semantics is a useful preliminary to reading the Ada Reference Manual.

Each instruction unit consists of a concept map followed by a sequence of frames. The concept map serves as a specification of the instruction unit, stating its purpose and indicating the logical relation among the concepts it introduces. It plays a dual role as a preliminary overview for students and a guide to authors in the design and implementation of the unit.

The development of this material was motivated by the desire to explore a new writing style appropriate for "computer textbooks". We hope eventually to expand this material into a computer-managed, educational database with hundreds of frames, multiple entry points, and multiple modes of traversal, so that it can be used by students with different backgrounds and rates of learning. Programs will eventually be under the control of an Ada compiler so the user can interact with them on line. The educational database could eventually be part of an Ada environment that supports both learning and substantive application programming.

Review of this material by Richard Bolz and his students at the Air Force Academy, and by students in the author's programming language class resulted in many improvements. Readers' comments and suggestions for further improvement and expansion of this material will be welcomed.

[1] Booch Grady, Software Engineering in Ada, Benjamin Cummings, 1982.

[2] Wegner Peter, Ada Self-Assessment Procedure, Communications of the ACM, October 1981.

* This work was supported in part by the Ada Joint Program Office through ONR contract #N00014-78C-0656.

DTIC ELECTE NOV 2 7 1984 S D A

() DRAFT: 10 October 1983

This document has been approved for public release and sale; its distribution is unlimited.

Table of Contents (Frames in Each Instruction Unit)

* Starred sections may be omitted on first reading

Accession For
NTIS [illegible]
FL-88 on file
A-1

1.0. Warming-Up Exercise

Ada is a programming language developed at the initiative of the Department of Defense for writing large application programs.

Its modular programming facilities are richer than those of Fortran or Pascal, so that large programs may be systematically constructed out of modular components.

But its expressions and statements are similar to those of Fortran, and its subprogram and datatype mechanisms are similar to those of Pascal.

Thus the reader familiar with Fortran or Pascal should have no difficulty in understanding corresponding features of Ada.

Programs in Ada are represented by character strings over an alphabet that includes letters, digits, and punctuation symbols. Characters may be combined into **lexical elements** of the following kinds:

Example 1: Lexical Elements of Ada

identifiers such as X, ALPHA, which serve as names of program entities.
literals such as 3, 3.5, which represent values.
delimiters, which include operators, punctuation symbols, and parentheses.
comments, which serve to document the program.

Lexical elements are the **atomic** program constituents of Ada. They can be combined to form expressions, statements, and other program structures.

"X + 3" is an expression consisting of the identifier "X", followed by the operator "+" followed by the literal "3".

"BETA := ALPHA + 365;" is an assignment statement which assigns the value of the expression "ALPHA + 365" to the identifier BETA.

In order to write Ada programs we must understand the rules for constructing programs out of component character strings.

The rules for building program structures from component character strings are called rules of **syntax**, while the rules for determining the result of program execution are called rules of **semantics**. We shall be concerned with both the syntax and the semantics of Ada programs.

Example: The syntax of expressions are the rules that determine how expressions are constructed from constituents. The semantics of expressions are the rules that determine how expressions are evaluated.

Which of the following assertions are false?
a) Ada supports a richer set of program modules than Fortran
b) Lexical elements are the atomic program constituents of Ada.
c) Expressions may include identifiers, literals, and delimiters.
d) Rules for building program structures out of character strings are called rules of semantics.
e) Rules of expression evaluation are rules of semantics.

Concept Map for Unit 1: Lexical Elements

The purpose of this instruction unit is to classify and describe the lexical elements out of which Ada programs are constructed. The relation among the various classes of lexical elements of Ada is indicated by the following concept map.

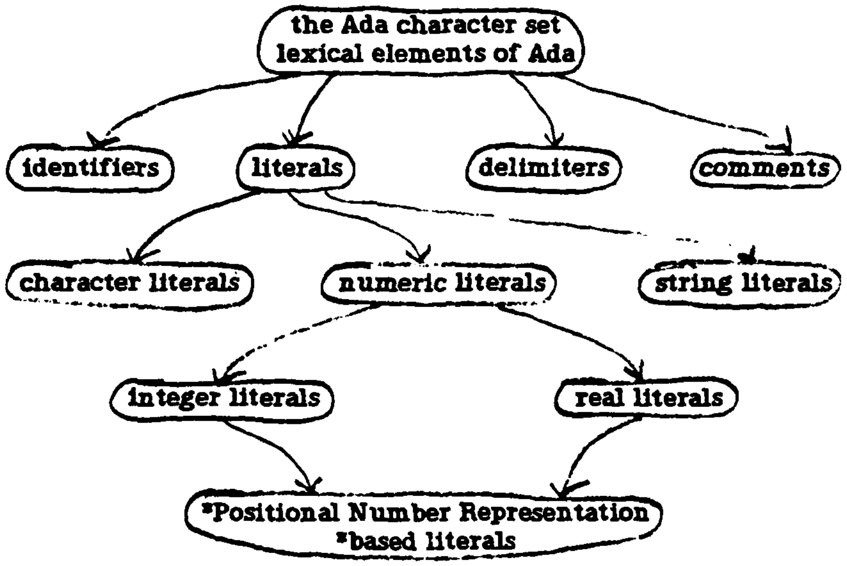


Figure 1: Concept Map for Lexical elements

This concept map serves as a guide to the development of the set of frames in the present instruction unit.

The first frame enumerates the Ada characters, and classifies them into subclasses such as digits, letters, and special characters.

In the second frame we examine a line of Ada code that contains identifiers, literals, delimiters, and comments.

The syntactic form and semantic role of identifiers, literals, delimiters, and comments is discussed in detail in subsequent frames.

As is evident from the concept map, literals are discussed in greater detail than other classes of lexical elements.

Subclasses of literals are defined, including numeric, character, and string literals. Numeric literals are further subdivided into integer literals and real literals.

The material on number representations and based literals is starred to indicate its greater level of difficulty. It is represented in the body of the instruction unit by two starred frames that may be omitted on first reading.

Note: A concept map represents static logical relations among concepts. The associated sequence of frames describes the concepts of the concept map dynamically. The order in which concepts are introduced does not necessarily correspond to a top-down traversal of the concept map, and the correspondence between frames and nodes of the concept map is not necessarily one-to -one.

1.1. The Ada Character Set

The Ada character set has 95 characters, including 52 upper and lower case letters, 10 digits, and the space character. In addition, there are 32 special characters that include operator symbols such as "+", parenthesis symbols such as "(", and punctuation symbols such as ";".

Example 2: The Ada Character Set

upper-case letters:
A B C D E F G H I J K L M N O P Q R S T U V W X Y

lower-case letters:
a b c d e f g h i j k l m n o p q r s t u v w x y z

digits:
0 1 2 3 4 5 6 7 8 9

the space character

special characters:

```
~ # & ' ( ) * + , - . / : ; < = > _|
! $ % ? @ [ ] ^ ' { }
```

This character set is the standard ASCII character set of the American National Standards Institute (ANSI), and is used for most current programming languages.

We shall consider both the syntactic rules for building program structures over this character set, and the semantic rules for associating meanings with program constituents.

In Ada the atomic program constituents are called **lexical elements** and include **identifiers** such as "ALPHA", **literals** such as "365", and **delimiters** such as "+" or ":=".

Identifiers, literals and delimiters can be combined into expressions such as "ALPHA+365" that determine rules for computing a value, and assignment statements such as "BETA:=ALPHA+365;" that record the value of expressions for later use.

Note on standards and standards organizations:
The American National Standards Institute (ANSI) and the International Standards Organization (ISO) promulgate standards for both character sets and programming languages. ASCII, which stands for "American Standard Code for Information Interchange", is the standard Ada character set. Ada became an approved ANSI standard language in February 1983, and will be considered for ISO standardization in the next year or two.

Which of the following assertions are false?
a) The character string "Melissa" contains seven letters.
b) Ada programs are composed of sequences of characters over the Ada alphabet.
c) The special characters of Ada include four styles of matching parentheses.
d) The character string "3.1416" contains five characters.
e) ASCII and Ada are ANSI standards.

1.2. Lexical Elements and Separators

The line of Ada text in the example below consists of seven lexical elements. Two are identifiers, one is a literal, three are delimiters, and one is a comment.

Example 3: Identifiers, Literals, Comments, and Delimiters

```
X := X + 1;          -- add 1 to current value of X
```

The seven lexical elements in this line include:
The identifier "X"
The delimiter ":="
A second instance of "X"
The delimiter "+"
The literal "1"
The delimiter ";"
The comment "-- add 1 to current value of X"

This line of Ada text includes space characters between its lexical elements. The space character is an example of a **separator**. The text of a program consists of a sequence of lexical elements and separators. But the effect of a program does not depend on its separators or comments but only on its identifiers, literals and delimiters.

The sequence of characters "X:=X+1;" with no spaces between lexical elements and no comments, has the same effect as the line of text in our example.

Our line of text is terminated by an (invisible) end of line symbol. The end of line symbol is an example of a **format effector**. Format effectors are separators which which have no external representation but are represented internally by a character code. They have an effect on the format of the output text rather than being preserved as characters in the output representation.

Separators and comments are designed to improve the appearance and readability of a program rather than affect the computational task it performs. We shall see later that readability is vitally important in determining the quality and costs of programming and cannot be ignored by the programmer. In writing a program we are concerned not only with communicating a sequence of instructions to the computer but also with communicating the structure and purpose of the program to humans who wish to debug or modify it. The fact that programs must be designed for communication with humans as well as computers has a profound impact on the task of programming.

Which of the following assertions are false?

a) The text of a program consists of a sequence of lexical elements and separators.
b) The effect of a program depends only on its lexical elements.
c) Extra spaces between lexical elements may change a program's effect.
d) Format effectors are separators which affect a program's output format.
e) Programs should be designed for communication with humans as well as computers.

1.3 Identifiers

Identifiers are represented by a string of one or more characters the first of which is a letter. The remaining characters, if any, may be letters or digits, as in X, X53. Isolated underscores may occur within identifiers to improve readability. Ada does not distinguish between upper- and lower-case letters in identifiers, so that X53 and x53 are always equivalent names for the same entity.

Identifiers serve as names of variables, procedures, types, or other entities. They should be chosen so that the name suggests the intended use.

Example 4: Examples of Identifiers

```
X              -- name for mathematical variable
LINE_COUNT     -- name for counting lines in a text editor
SQRT           -- identifier used to denote square root function
INTEGER        -- identifier that denotes a predefined type
if begin loop  -- three reserved identifiers separated by space symbols
```

Variables such as X are **programmer-defined identifiers**, and must be explicitly **declared** before they can be used for purposes of computation.

Programmer-defined identifiers may be contrasted with **predefined identifiers** that provide facilities on which the programmer can rely without having to define them. The predefined identifiers of Ada include the **reserved identifiers**.

Reserved identifiers serve to define primitive language constructs for purposes such as conditional branching, iteration, and program structuring. Ada has 63 **reserved identifiers** including **if**, **loop**, **begin**, **procedure** with a fixed language-defined meaning that cannot be redefined by the programmer.

Identifiers such as INTEGER have a predefined meaning which may be changed if there is good reason to do so. INTEGER determines properties of integers such as their maximum size which may require redefinition when there is a change in the computer word size for integers.

Identifiers such as SQRT, which are defined as subprograms of a program library, have a status between that of programmer-defined and predefined identifiers. They are programmer-defined identifiers from the point of view of the programmer who defines the library program but predefined from the point of view of the user of the library program.

Identifiers may be viewed as names of **computational resources**, including **data resources** such as variables and **program resources** such as procedures. Predefined identifiers name resources provided by the language, programmer-defined identifiers name resources defined by the programmer, and library identifiers name resources provided by the program library.

Which of the following assertions are false?
a) The first character of an identifier must be a letter.
b) The identifiers XYZ and xyz always denote the same entity.
c) Programmer-defined identifiers must be declared before they can be used.
d) Predefined identifiers cannot be redefined by the user.
e) Reserved identifiers denote **language-defined** computational resources.

1.4. Numeric Literals

Literals are lexical units that represent a value. Numeric literals represent numeric values. The numeric values representable in Ada include integers, which are exact values, and real numbers, which are approximate values.

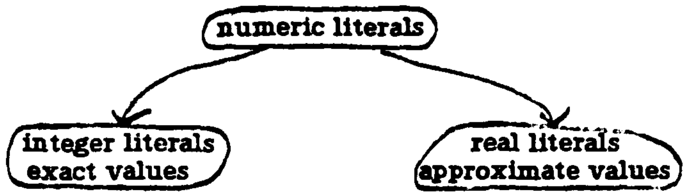


Figure 2: Exact and Approximate Numeric Literals

Integers consist of a numeric part optionally followed by an exponent. The numeric part consists of a sequence of digits possibly separated by underscores.

Example 5: Integer Literals (represent exact values)

```
5               -- represents the integer five
565             -- five hundred sixty-five
1_000_000       -- value is 1 million, underscores improve readability
1E6             -- alternative representation of 1 million
```

A numeric value may in general be represented in many different ways by a numeric literal. 1_000_000 and 1E6 are two different ways of representing the integer "one million" by a literal.

Real literals, like integer literals, consist of a numeric par optionally followed by an exponent. They contain a decimal point in their nun eric part.

Example 6: Real Literals (represent approximate values)

```
0.5            -- one half
0.0            -- zero
3.1416         -- approximate value of pi
1000.0         -- one thousand
1.0E3          -- positive exponent, value 1000.0
1.0E-3         -- negative exponent, value .001
```

Which of the following assertions are false?

a) 1_000_000_000 represents the integer one billion.
b) 1.0E9 represents the real number one billion.
c) An integer is an exact value and a real number is an approximate value.
d) 1.0E-4 is less than 1.0E-5.
e) Underscores are significant in identifiers but not in numeric literals.

1.5. Character and String Literals

A character literal is represented by a character in single quotes.

Example 7: Character Literals

```
'a'     -- the character literal a
'+'     -- the character literal +
'''     -- the character literal '
```

A string literal is represented by a string of zero or more characters in double quotes. The number of characters in a string literal is said to be its length.

The double quote symbol cannot normally appear within a string since it is interpreted as the string terminator. In order to get around this restriction we represent occurrences of " within a string by two double quote symbols.

Example 8: String Literals

```
"a"         -- the string literal a of length one
"ABC"       -- the string literal ABC of length three
""          -- the empty string of length zero
"""A"""     -- the string literal "A" of length three
```

The character literal 'a' and the string literal "a" of length one are not equivalent. Character and string literals represent values of different types.

Small and capital letters are not equivalent within a quoted string. Thus 'a' and 'A' are not equivalent. Likewise "ABC" and "abc" are not equivalent, although the strings ABC and abc are equivalent as identifiers.

Note that there is essentially only one way of representing a character string by a string literal. This contrasts with the multiplicity of ways of representing numbers by numeric literals.

Which of the following assertions are false?

a) 'x' and 'X' are two different character literals.
b) "x" and "X" are two different string literals.
c) """John""" represents the literal "John".
d) "I think therefore I am" is a string literal with 18 characters.
e) Character values have a unique external representation as character literals, while numeric values may be represented by a variety of different equivalent numeric literals.

1.6. Comments

A comment in Ada starts with a double hyphen and is terminated by an end of line.

Comments can appear either in a line following a statement or declaration, or as stand-alone comments occupying a complete line.

Example 9: Attached and Stand-Alone Comments

```
X := X + 1;   -- increase value of X by 1


-- comments can appear on any line
-- following all lexical units of the line
-- or they can appear as stand-alone
-- comments taking up the whole line
-- like the present five-line comment
```

Comments do not affect the computation performed by a program but can greatly increase the readability, modifiability, and understandability of programs. They play no role in communicating with the computer but play a key role in communicating the purpose and structure of the program to other humans. They should not be viewed as a "cosmetic" add-on, but as an integral part of the overall programming process that reduces the cost of debugging, maintenance and other operations that require understanding of the program by humans.

Which of the following assertions are false?

a) Comments can appear on the same line as an executable statement provided they appear after the executable statement.
b) Removing all comments from a program does not affect the computation performed by the program.
c) Comments are useful for program development but should be removed when testing has been completed so as to improve run-time efficiency.
d) Comments are a form of documentation that may greatly enhance the understandability of a program.
e) Good comments can greatly reduce the costs of developing and maintaining a program.

* 1.7. Positional Number Representation

The decimal number "986" has a 6 in the units position, an 8 in the tens position, and a 9 in the hundreds position. Its value is given by:

986 = 9*100 + 8*10 + 6

This notation for decimal number representation is called a **positional representation** because the value of each digit depends on its position in the digit string. The two salient features of the positional notation for decimal numbers are:

a) Decimal numbers consist of a sequence of decimal digits taken from the set {0,1,2,3,4,5,6,7,8,9}.
b) The rightmost position is the units position and each position to the left is worth a factor of **ten** more than its right neighbor.

Decimal notation has become the standard representation in part because we use our ten fingers for counting. Note that the term digit is the Latin word for finger.

Binary numbers are an alternative form of number representation particularly suited to computers. The two salient features of binary number notation are the following:

a) Binary numbers consist of a sequence of binary digits taken from the set {0, 1}.
b) The rightmost position is the units position and each digit position to the left is worth a factor of two more than its right neighbor.

Thus the binary number 1101 has a 1 in the units position, a 0 in the twos position, a 1 in the fours position, and a 1 in the eights position.

1101 = 1*8 + 1*4 + 0*2 + 1*1 = 8 + 4 + 1 = 13

Our definitions of salient features of decimal and binary numbers suggest the following characterization of numbers of an arbitrary number base **N**.

a) Base **N** numbers consist of a sequence of digits from the set {0,1,...,N-1}.
b) The rightmost position is the units position and each digit position to the left is worth a factor of **N** more than its right neighbor.

Example: Base 3 numbers are sequences of digits taken from the set {0,1,2}. The base 3 number 212 has the following value.

212 = 2*9 + 1*3 +2*1 = 18 + 3 + 2 = 23

Which of the following assertions are false?
a) The decimal number 756 has the value 7*100 + 5*10 + 6.
b) In positional number representations each digit position to the right is worth a factor of two more than its left neighbor.
c) The binary number 1011 has the decimal value 11.
d) The base 3 number 1011 has the decimal value 31.
e) The base 4 number 1011 has the decimal value 69.

* 1.8. Based Literals

Ada allows numbers to be represented as **based literals** in any number base between two and sixteen.

Based literals consist of a base specified in decimal, followed by a number in the specified base enclosed by the symbol #, followed optionally be a decimal exponent.

Example 10: Based Binary and Ternary Literals

```
2 #111#          -- base 2 (binary) literal with value 4 + 2 + 1 = 7
2 #111# E3       -- base 2 literal with value 7*8 = 56
3 #212#          -- base 3 (ternary) literal with value 18 + 3 + 2 = 23
3 #212# E1       -- base 3 literal with value 23*3 = 69
```

Based literals with base 16 are called hexadecimal literals. The digits 10,11,12,13,14,15 of a hexadecimal literal are represented by the letters A,B,C,D,E,F.

Example 11: Hexadecimal Literals

```
16 #A1# = 10*16 + 1 = 161
16 #BF# = 11*16 + 15 = 191
16 #FF# = 15*16 + 15 = 255
```

Based literals may be represented not only by integers but also by real numbers. Based literals representing real numbers have a decimal point in their number part.

Example 12: Based Real Literals

```
2 #1.11#               -- one and three quarters
3 #1.11#               -- one and four ninths
2 #1.11# E1            -- Three and a half
2 #1111_1111.# E-4     -- fifteen and fifteen sixteenths
16 #FF.0# E-1          -- fifteen and fifteen sixteenths
```

Which of the following assertions are false?

a) The based literal 3#222# and the decimal literal 26 represent the same value.
b) 4#333# and 63 are two alternative representations of the same value.
c) 15#EF# represents the value 225.
d) 8#70.0#E-1 represents the integer 7.
e) 5#44.0#E-1 represents the number four and four fifths.

Concept Map for Unit 2: Syntactic Notation

The purpose of this unit is to introduce a notation for defining the syntax of programming language constructs. The relation among concepts introduced in this unit is given by the following concept map

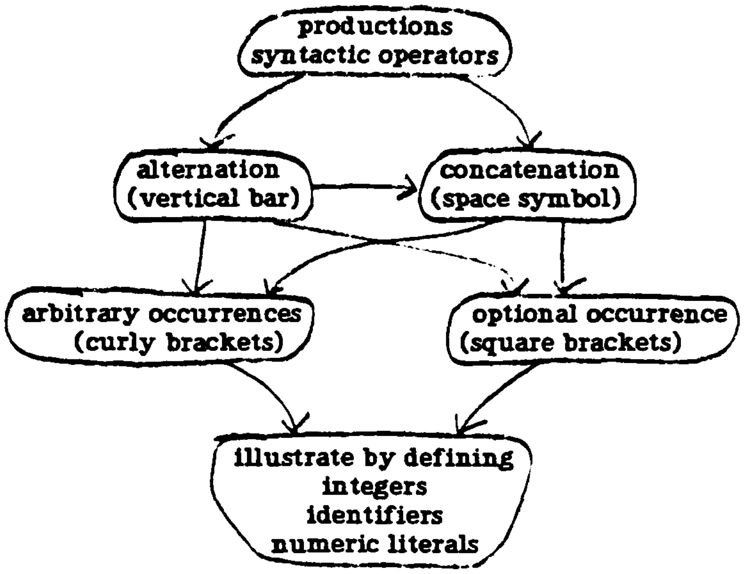


Figure 3: Concept Map for Syntactic Notation

Syntactic categories in programming languages correspond to parts of speech in natural language. Identifiers and literals correspond to nouns, while operators correspond to verbs.

Productions are a mechanism for naming new syntactic categories constructed from previously defined syntactic categories. There are four **syntactic operators** for constructing new syntactic categories.

(1) The alternation operator X | Y, which constructs the union of the character strings X and Y.

(2) The concatenation operator X Y, which determines the set of all character strings consisting of an instance of X followed by an instance of Y.

(3) The curly bracket { X }, which specifies an arbitrary number of occurrences of instances of the syntactic category X.

(4) The square bracket [X], which specifies the optional occurrence of an instance of X. specifies optional occurrence of a construct.

The use of productions to define syntactic categories is illustrated by defining the syntactic categories **integer**, **identifier** and **numeric literal**.

The first frame introduces productions and the alternation operator. The second frame introduces the concatenation operator. The third frame introduces arbitrary and optional occurrence and develops the syntactic definition of identifiers. The fourth frame develops the syntactic definition of numeric literals.

2.1. Productions and Syntactic Categories

The syntax of Ada is defined by productions that serve to name syntactic categories.

Productions have a left-hand-side specifying the category being defined and a right-hand-side which specifies the definition.

defined-syntactic-category ::= syntactic-definition

The syntactic category "**digit**" may be defined by the following production:

digit ::= 0|1|2|3|4|5|6|7|8|9

The symbol **:=** may be read as "**is defined as**". The vertical bar may be read as "or". The production for **digit** may therefore be read as:

digit is defined as 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9

The vertical bar is called the **alternation operator**. It is used to define new syntactic categories in terms of a set of alternatives which constitute instances of that category.

The syntactic categories "**upper_case_letter**" and "**lower_case_letter**" may be defined in terms of characters of the Ada alphabet as follows:

upper_case_letter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
lower_case_letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

The syntactic categories "**letter**", and "**letter_or_digit**" may be defined in terms of previously-defined syntactic categories as follows:

letter ::= upper_case_letter | lower_case_letter
letter_or_digit ::= letter | digit

The alternatives of an alternation operator can be character strings (terminals) or syntactic categories.

Which of the following assertions are false?

a) The syntactic category "**letter**" determines a set with 52 members.
b) **letter_or_digit** is the union of the set of letters and the set of digits.
c) The alternatives of an alternation operator may be syntactic categories.
d) The categories **digit** and **letter** determine disjoint subsets of the Ada character set.
e) The number of elements in the syntactic category "**X|Y**" is the sum of the number of elements in the categories X and Y.

2.2. The Concatenation Operator

Definition: The concatenation "X Y" of two character strings "X" and "Y" is the character string consisting of the string X followed by the string Y.

The concatenation operator is denoted by a space. It may be applied to two characters, as in "a b" to yield the two-character sequence "ab". It may be applied to two character strings, as in "abc xyz" to yield the character string "abcxyz".

The idea of concatenation may be extended so that we can concatenate not only character strings but also syntactic categories.

Definition: The concatenation "X Y" of two syntactic categories "X" and "Y" is the set of all character strings consisting of a string in the set X followed by a string in the set Y.

Example 13: Concatenation of Syntactic Categories

```
X ::= A|B       -- X is a set with two letters {A, B}
Y ::= 1|2       -- Y is a set with two digits {1, 2}
XY ::= X Y      -- XY is a set with four strings {A1, A2, B1, B2}
```

The set of 100 two-digit strings and the set of 1000 three-digit strings may be defined as follows:

```
two_digits ::= digit digit             -- set of 100 two-digit strings
three_digits ::= two_digits digit      -- set of 1000 three-digit strings
```

The combined use of the alternation and concatenation operators provide a powerful mechanism for defining sets of character strings containing large numbers of elements.

In particular, if X is a syntactic category with M alternative forms and Y is a syntactic category with N alternative forms, then X Y is a syntactic category with M*N alternative forms.

Which of the following assertions are false?

a) Syntactic categories determine sets of character strings.
b) The number of elements in the set "letter digit" is the product of the number of elements in the sets "letter" and "digit".
c) The set of character strings determined by "X Y" is always bigger than the set of character strings in X or Y.
d) three_digits digit and two_digits two_digits denote identical sets of character strings.
e) Productions serve to associate a name with a set of character strings.

2.3. The Syntax of Integers and Identifiers

Arbitrary replication of syntactic constructs is indicated by curly brackets.

{ a }   -- set containing zero or more instances of the letter "a"
{ digit }   -- set with an arbitrary number of (zero or more) digits

Curly brackets allow us to define sets with an infinite number of characters. The infinite set of unsigned integers may be defined as follows:

unsigned-integer ::= digit { digit }   -- includes 3, 35, 356

Square brackets denote an optional occurrence of the enclosed syntactic construct. For example the optional occurrence of underscores in integer literals is captured by the following syntactic definition:

integer ::= digit { [_] digit } -- includes 3, 3_000, 1_000_000

This definition reads: An integer is defined as a digit followed by an arbitrary number of digits each of which may be optionally preceded by an underscore.

The syntactic definition of identifiers is similar in structure to the above definition of the class of integers.

identifier ::= letter { [_] letter_or_digit} -- includes X, X56, X_56

"An identifier in Ada consists of a letter followed by an arbitrary number (zero or more) of occurrences of a letter or digit each of which may be optionally preceded by an underscore."

The syntactic definitions above use all four of our mechanisms for syntactic definition.

alternation, denoted by the vertical bar
concatenation, denoted by the space symbol
arbitrary replication, denoted by curly brackets
optional occurrence, denoted by square brackets

Alternation is used in defining the categories **letter** and **digit**. Concatenation is used in specifying that an identifier is an element of the set **letter** concatenated with an element of the set { [_] **letter_or_digit** }. The curly and square brackets are used in the construct { [X] Y } for specifying an arbitrary number of instances of a category Y each of which may be optionally preceded by an instance of the category X.

Which of the following assertions are false?
a) Curly brackets allow sets with an infinite number of elements to be defined.
b) The production "**digit_string ::= digit {digit}**" asserts that every string of one or more digits is an instance of the syntactic category "**digit_string**".
c) The production "**integer ::= digit { [_] digit}**" defines an "integer" to be a digit string which may have embedded single underscores.
d) The production "**abc-string ::= {a|b|c}**" asserts that a three-character string over the alphabet a,b,c is an instance of "**abc-string**".
e) The set of strings which are legal identifiers is a subset of the set of all possible strings over the alphabet of Ada.

* 2.4. The Syntax of Numeric Literals

The basic mechanisms for syntactic definition have been illustrated in previous frames. These mechanisms are further illustrated below by developing a syntactic definition for numeric literals.

Numeric literals may be decimal literals or based literals.

```
numeric_literal ::= decimal_literal | based_literal
```

Decimal literals consist of a numerical part followed optionally by an exponent. The numeric part may be a digit string (for integers) or two digit strings separated by a decimal point (for reals). The exponent part consists of the letter E followed by a digit string optionally preceded by a sign.

```
decimal_literal ::= number_part [exponent]
number_part ::= digit_string [. digit_string]
digit_string ::= digit { [-] digit}
exponent ::= E [sign] digit_string
sign ::= + | -
```

Based literals consist of a base which is an integer in the range 2 through 16, a number part in an extended alphabet that may include the characters A B C D E F, and an optional decimal exponent.

```
based _literal ::= base #based_number# [exponent]
base ::= 2|3|4|5|6|7|8|9|10|11|12|13|14|15
based_number ::= extended_digit_string [. extended_digit_string]
extended_digit_string ::= extended_digit { [_] extended_digit}
extended_digit ::= digit|A|B|C|D|E|F
```

The above syntactic definitions express the relatively complex rules for decimal and based literals in a concise, but understandable way.

It should, however, be noted that the set of based literals defined here contains certain strings that we would prefer to exclude, such as "2#58#" and "10#FF#".

Refinement of our definition so that the digits of a based number are restricted to be smaller than the base would make it much longer. We could define separate syntactic categories for binary-digit, ternary-digit etc, and syntactic categories for based-binary-number, based-ternary-number etc. This would require 30 extra productions, but would result in a more precise characterization of the set of based literals.

Which of the following assertions are false? a) The number part of a decimal literal need not contain a decimal point.
b) The decimal digits are a subset of the extended digits.
c) Exponents for decimal literals and based literals have the same syntactic form.
d) The value of the based literal "2#FF#" is less than that of the based literal "3#FF#".
e) All values specifiable by based literals can also be specified by decimal literals.

Concept Map for Unit 3: Expressions, Statements, and Declarations

The purpose of this unit is to introduce the basic ideas of expressions, statements, and declarations, and to show how declarations and statements may be combined into simple program structures.

The relation among these concepts is illustrated by the following concept map.

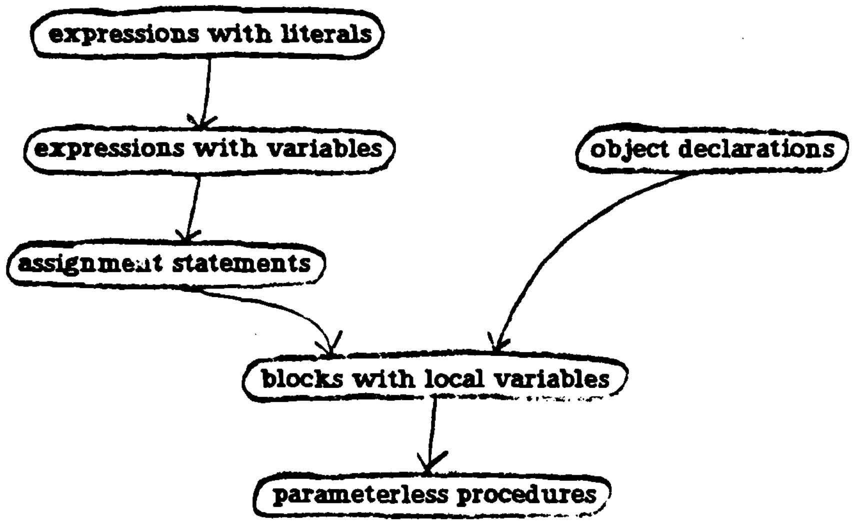

expressions with literals
expressions with variables
object declarations
assignment statements
blocks with local variables
parameterless procedures

Figure 4: Concept Map for Expressions, Statements, and Declarations

Expressions are rules for computing a value. Frame 1 introduces expressions with literals and the idea of operator precedence. Frame 2 introduces expressions with variables.

Assignment statements allow results computed by expressions to be recorded as values of variables. Frame 3 introduces assignment statements and shows how the value recorded by one assignment statement can be used in subsequent statements.

Object declarations serve to introduce programmer-defined identifiers and describe their properties. Frame 4 shows that object declarations of INTEGER variables determine the value-set and applicable operations of declared variables, and indicates that declarations result in storage allocation for variables at the time of their elaboration.

Frames 5 and 6 introduce two program structures (blocks and procedures) for declaring a set of variables and then using them for computation. Procedures name groups of declarations and statements so that they can be subsequently called by a procedure call. Blocks are anonymous and must be executed in-line at the point of the program where they occur.

Frame 7 reviews the syntax of procedure declarations, and contrasts the declarative nature of declarations with the imperative nature of assignment statements.

3.1. Integer-Valued Expressions

Expressions are rules for computing a value by the application of operators to operands.

The rules for evaluating arithmetic expressions in Ada are similar to those of most other programming languages. These rules are illustrated below for integer-valued expressions with literal operands.

Example 14: Expressions that Compute Integer Values

```
3                  -- expression with value 3
3 + 4              -- expression with value 7
(3 + 4) * 5        -- value is 35
3 + 4 * 5          -- value is 23, 3+(4*5)
```

The value of the expression containing just the integer literal 3 is simply the value of the literal.

The value of the expression "3+4" is computed by applying the addition operator to the values of "3" and "4", yielding a value which may be represented by the literal "7".

The expression "(3+4)*5" contains parentheses which indicate the order in which operators are to be applied to operands. (3+4) is evaluated first, and the resulting value is used as an operand of *, yielding the value 7*5 = 35

The expression 3+4*5 does not contain parentheses to indicate the order of evaluation. However, by convention, multiplication has precedence over addition, so that there are implicit parentheses 3+(4*5) which determine a value 3 + 20 = 23.

Rules for operator precedence allow the parentheses in the expression 3 + (4 * 5) to be removed, making it shorter and more readable, but require parentheses in (3 + 4) * 5 since their removal would change the value of the expression.

The exponentiation operator, denoted by ** has precedence over both addition and multiplication, so that

```
5 + 4 * 3 ** 2 = = 5 + (4 * (3 ** 2)) = 5 + 4 * 9 = 5 + 36 = 41
```

The operators +, *, ** are said to be **binary operators** because they take two operands. The binary operators also include minus(-) with the same precedence as + and divide(/) with the same precedence as *. Thus "3 - 10 / 5 = 3 - 2 = 1"

Operators with just a single operand are said to be **unary operators**. The operators +,- may be used as unary operators as in "2**(-3)".

Which of the following assertions are false?

a) The expression "3*4+5*6" has the value 42.
b) "3*4+5*6" has the same value as (3*4)+(5*6).
c) Operator precedence rules allow the parentheses in "(4*3)**2" to be omitted without changing the value of the expression.
d) * is a binary operator which takes two integer operands and yields an integer result
e) The operator - may be used both as a binary and as a unary operator.

3.2. Expressions with Variables

Literals have fixed values which can be determined from their syntactic form. Variables have values which may be updated by assignment during the course of a computation. At any given point of a computation the current value of a variable is that which has most recently been assigned to it. The value of an expression depends on the current values of its variables.

Example 15: Integer Expressions with Variables

```
Let I = 3, J = 4, K = 5, be the current values of the variables I,J,K

I                      -- expression with value 3
I + J                  -- expression with value 3 + 4 = 7
(I + J) * K            -- expression with value 7 * 5 = 35
I + J * K              -- expression with value 3 + 20 = 23
K + J * I ** 2         -- expression with value 5 + 4 * 3 ** 2 = 41
```

The expression consisting of the single variable I returns the current value of I as its value.

Evaluation of the expression I+J is similar to evaluation of 3+4. Both are evaluated by first evaluating the operands and then applying + to the resulting values. The principal difference is that the values of the operands I,J in the expression I+J can vary during execution, while the values of the operands 3,4 in the expression 3+4 are fixed, so that the expression could be evaluated prior to execution.

The expression "(I + J) * K" is evaluated by first evaluating "(I + J)" and then using this value as an argument of *, yielding the value 7 * 5 = 35.

In the case of "I + J * K" the operator * operates on the values of J and K, and the operator + operates on the values of I and J * K, yielding the value 3+20 = 23.

The expression K+J*I**2 is evaluated as though it were parenthesized K+(J*(I**2)), yielding the value 41.

We assumed above that all variables had current values without indicating how these values were assigned to them. The primary method of assigning values to variables is considered in the next frame.

Which of the following assertions are false?

a) The value of a variable may be modified during program execution.
b) "+" can take literals, variables, and expressions as its operands.
c) The value of an expression depends on the current values of its variables.
d) When I=3, J=4, K=5, then I + J * K ** 2 has the value 103.
e) I**J and J**I have the same value.

3.3. Assignment Statements and Memory Cells

Assignment statements provide a mechanism for assigning the computed value of an expression to a variable. The left-hand-side of an assignment statement contains the variable to which a value is assigned, while the right-hand-side contains the expression whose value is being computed.

Example 16: Assignment Statements

```
I := 3;           -- assign 3 to I
J := I + 1;       -- assign 4 to J, using previous value of I
K := I + 2 * J;   -- assign 11 to K, using values of I and J
```

Values assigned to a variable in a given assignment statement may be used in subsequently executed assignment statements. The assignment statement for J uses the previously assigned value of I while the assignment statement for K uses the previously assigned values of both I and J.

Assignment to a variable destroys the previous value of that variable so that it is lost for ever. This property of assignment reflects the property of computer memory cells that storage of a value in a memory cell destroys the previous value.

Variables in Ada, as in other higher-level languages, model the properties of memory cells. Occurrence of a variable I on the right-hand-side of an assignment statement is modelled by a LOAD I instruction which loads its value into a register of the central processing unit. Occurrence of a variable J on the left-hand-side of an assignment statement is modelled by a STORE J instruction.

The assignment statement J := I + 1; can be implemented by the following machine language style instructions:

```
LOAD  from memory cell I
ADD  value of literal 1
STORE  in memory cell J
```

Higher-level languages like Ada allow us to replace such machine language instructions by user-friendly abstractions like "J := I + 1;

The relation between variables and memory cells is not accidental. It reflects the structure of the underlying computers for which higher-level languages were designed, which has not changed since the days of Fortran.

Which of the following assertions are false?

a) Assignment statements allow values of variables to be changed.
b) Assignment to a variable destroys its previous value.
c) The assignment operator models storage of a value in the computer memory.
d) Occurrence of a variable on the right hand-side of an assignment statement models loading of a value from the memory into the central processing unit.
e) "I := I + 1;" replaces the integer value of I by its successor.

3.4. Object Declarations

Declarations provide a mechanism for the naming of program entities. Object declarations are a particular class of declarations for naming variables. The object declaration below specifies that I is a variable of the type INTEGER.

Example 17: Object Declaration for the Variable I

```
I : INTEGER;    -- declare variable named I of the type INTEGER
```

Variables in Ada must be declared to have a type compatible with the operators that will subsequently be applied to them. The object declaration "I: INTEGER;" associates with I an object that can have values of the type INTEGER and can be operated on by operations that expect operands of the type INTEGER. The object associated with I may be thought of as a memory cell specialized to hold values of the type INTEGER.

The following object declaration declares three variables I, J, K of the type INTEGER.

```
I, J, K: INTEGER;    -- declare three integer variables named I,J,K
```

Its execution creates three objects (memory cells) to which values of the type INTEGER can be assigned.

Declarations are said to be **elaborated** when they are encountered during program execution. Elaboration of the declaration "I: INTEGER;" causes the integer-valued variable I to come into existence. If we think of the variable I as a memory cell, then elaboration of the declaration corresponds to allocation of the memory cell. Object declarations are normally implemented by allocation of a memory cell for each declared variable.

Declarations of variables impose constraints on the way in which the variables may be used in subsequent computations. The declaration "I: INTEGER;" restricts the variable I so that it can take only integer values and can be operated on only by operations that expect integer-valued operands.

The type of a variable describes the set of values it can assume and the kinds of operators which can be applied to its values. Variables of the type INTEGER can take integers as their values and can appear as operands of operators that expect integer values as their arguments.

Which of the following assertions are false?

a) Declaration of the variables I,J,K must occur prior to their use.
b) The declaration of a variable always specifies its initial value.
c) The value assigned to a variable must be compatible with the type specified in its type declaration
d) Declaration of a variable corresponds to allocation of a memory cell in which values of that variable may be stored.
e) The type of a variable determines its value set and set of applicable operations.

3.5. Blocks with Local Variables

Program structures may contain both **object declarations** for variables and **statements** which specify computations on declared variables. The simplest such program structure is the **block**.

Blocks have a declarative part introduced by the reserved identifier **declare** and a statement part introduced by the reserved identifier **begin** and terminated by the reserved identifier **end**.

Example 18: Block with Three Assignment Statements

```
declare               -- keyword introducing declarative part of block
 I,J,K: INTEGER;      -- declarations of local variables I, J, K
begin                 -- keyword introducing statement part of block
 I := 3;              -- assign 3 to I
 J := I + 1;          -- assign 4 to J
 K := I + 2 * J;      -- assign 11 to K
end;                  -- keyword terminating the block
```

Elaboration of the declaration "I,J,K: INTEGER;" causes the variables I,J,K to come into existence so that they are available for use in executing the statements of the block. When execution of the block is completed the variables declared on entry become inaccessible. We may think of variables declared in a block as being dynamically allocated on entry to the block and deallocated on exit from the block.

Variables declared in the declarative part of a block are said to be **local** to the block. They are known only in the statement part of the block and cannot be used outside the textual boundaries of the block.

The textual range over which a declared variable is known is called the **scope** of the variable. The scope of variables I, J, K extends from their point of declaration to the end of the block in which they are declared.

The block above is an **anonymous** program structure which must be embedded in a textually enclosing program structure before it can be executed. In the next frame we shall introduce **named** program structures (procedures) which can be invoked by procedure calls.

Which of the following assertions are false?

a) A block consists of a declarative part containing declarations of local variables, followed by a statement part containing statements that may perform computations on local variables.
b) The keyword **declare** introduces the declarative part of the block.
c) The local variables I,J,K are known only within the block and cannot be used in statements outside the scope of the block.
d) Only variables declared in the declaration part of a block may be used in its statement part.
e) The type of I,J,K specified in the declaration part is compatible with the values assigned to I,J,K in the statement part.

3.6. Parameterless Procedures

Procedures have a declarative part in which local identifiers may be declared and a statement part that may perform computations on locally declared identifiers. In this respect they have a structure similar to that of blocks.

Procedures differ from blocks in having a name that allows them to be explicitly called (invoked) when they are needed.

Example 19: Procedure declaration for ASSIGN

```
procedure ASSIGN is  -- declare a procedure named ASSIGN
  I,J,K: INTEGER;    -- declarations of local variables I, J, K
begin                -- keyword introducing statement part of procedure
  I := 3;            -- assign 3 to I
  J := I + 1;        -- assign 4 to J
  K := I + 2 * J;    -- assign 11 to K
end ASSIGN;          -- keyword terminating the procedure
```

This example of a procedure declaration serves to associate the declarations and statements of the procedure with the name **ASSIGN** in much the same way that the declaration "**I: INTEGER;**" associates an object which can take integer values with the name **I**. It may be invoked by the following procedure call.

```
ASSIGN;  -- invoke the previously declared procedure ASSIGN
```

This call causes the procedure to be executed, creating objects (memory cells) for the three local variables **I,J,K**, performing computations on local variables, and deleting the locally declared variables on exit from the procedure.

Procedure declarations serve to introduce identifiers that are names of procedures while object declarations serve to introduce identifiers that are names of variables. Procedures are executable sequences of actions and play a very different role in computation from objects which are place holders for storing data values. Ada's declarative mechanism for associating identifiers with program entities may be used both for entities that determine sequences of actions and for entities that represent data objects. The difference between action and data entities is reflected by restrictions in the way declared entities may be used. Thus data objects may be accessed and updated by assignment statements while procedures may be called and executed by procedure call statements.

Which of the following assertions are false?

a) Both blocks and procedures may have locally declared variables.
b) Procedures have a name that allows them to be called when they are needed, while blocks are anonymous and are executed in-line as embedded components of a larger computation.
c) The procedure **ASSIGN** has three parameters **I,J,K**.
d) The local variables **I,J,K** are created on entry to the procedure when it is called and become inaccessible when execution of the procedure is completed.
e) Since all assignments are to local variables, execution of this procedure cannot have any effect on any larger computation.

3.7. Review of Declarations and Statements

We have introduced expressions, which perform computations by applying operators to operands, assignment statements, which record the value of computed expressions, and object declarations, which serve to name and describe the properties of variables. Blocks and procedures are composite program structures that allow us to specify computations in terms of object declarations and statements that make use of declared variables.

Procedures and blocks have a declaration part that describes local entities that come into existence when the procedure is called and a statement part that specifies a sequence of actions on locally declared entities.

Example 20: Syntactic Structure of Simple Procedures

```
simple-procedure-declaration ::=  procedure NAME is
                                    declaration-part
                                  begin
                                    statement-part
                                  end
```

The previously-discussed **ASSIGN** procedure has a declaration part that declares three variables of the type **INTEGER** and a statement part that assigns values to these variables. It is simple in its structure and does not do anything that is particularly useful. But it illustrates in embryo form the underlying structure of more complex Ada programs.

Both the **declarative** mechanisms in Ada for **describing** computational entities and the **imperative** mechanisms for specifying sequences of actions are very rich. In the next few frames, additional declarative and imperative mechanisms of Ada are introduced to allow richer classes of computations to be specified.

First we shall increase our declarative repertoire, introducing the predefined types **FLOAT** and **BOOLEAN**, the idea of programmer-defined types, and a particular class of programmer-defined types called enumeration types. Then we shall expand our imperative repertoire by introducing the **if** statement for selection among alternative sequences of actions, the **loop** statement for the repetitive execution of statements. Functions which have input parameters whose value is supplied by the caller and return a value as output to the caller are introduced.

Which of the following assertions are false?

a) Declarations **describe** computational entities while statements perform actions on on computational entities.
b) Computational entities declared local to a procedure come into existence at compile time.
c) Declarations in the declaration part of a procedure are local to that procedure.
d) The procedure **ASSIGN** declares three integer objects in its declaration part and performs actions on them in its statement part.
e) The actions that may be performed in the statement part include the assignment of values to variables.

Concept Map for Unit 4: Scalar Data Types

The purpose of this unit is to introduce basic ideas relating to types. We restrict ourselves to scalar types (which have no components) and introduce structured types such as arrays in later instruction units.

The relation among concepts introduced in this section is as follows:

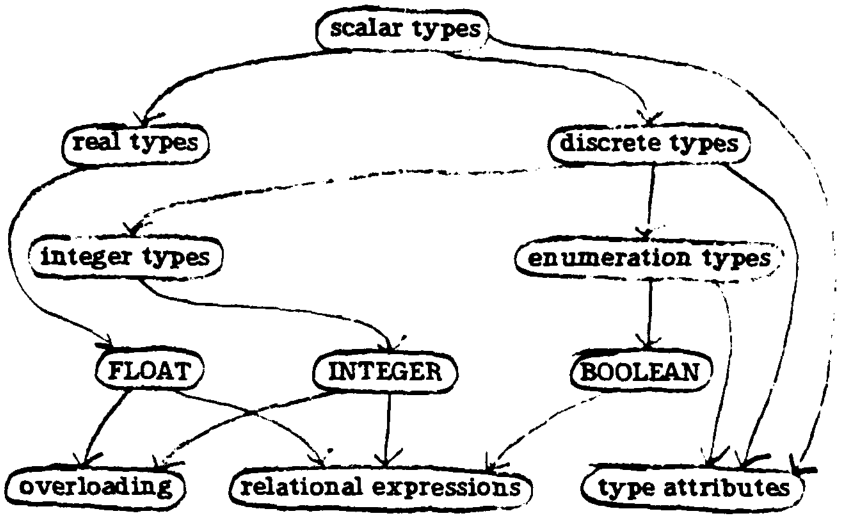


Figure 5: Concept Map for Scalar Types

The scalar types include the real types and the discrete types. The discrete types in turn include the integer types and enumeration types. The predefined types **FLOAT**, **INTEGER**, and **BOOLEAN** are respectively instances of real, integer, and enumeration types.

In introducing the subject of types we start with the familiar types **INTEGER**, **FLOAT** and **BOOLEAN**, and consider type classes such as the discrete types only after experience in using familiar types has been gained.

Since the type **INTEGER** has already been extensively used in previous examples we start with the type **FLOAT**. Frame 1 introduces floating point literals and variables, and discusses overloading of arithmetic operators for integer and floating-point types. Frame 2 introduces the type **BOOLEAN** and the use of **BOOLEAN** expressions to select alternatives in **if-then-else** statements. Frame 3 examines relational expressions. Frame 4 introduces type declarations for programmer-defined types. Frame 5 introduces enumeration types that enumerate their value set explicitly in the type definition. Frame 6 examines scalar types and introduces the notion of type attributes denoted by special composite identifiers of the form **TYPE'ATTRIBUTE**. Frame 7 introduces discrete types and examines type attributes of discrete types.

4.1. Floating Point Types

Floating point numbers are representable by floating point literals (containing a decimal point). The procedure FLOATING_POINT declares three variables of the predefined type FLOAT and assigns floating-point values to them.

Example 21: FLOATING_POINT Procedure

```
procedure FLOATING_POINT is       -- a procedure declaration
  X,Y,Z: FLOAT;                   -- which declares three floating-point variables
begin                             -- and has a statement sequence
  X := 3.5;                       -- which evaluates floating-point expressions
  Y := X + 1.0;                   -- and assigns their vales
  Z := X + 2.5 * Y;               -- to floating-point variables
end
```

Floating-point literals can be distinguished from integer literals because they contain a decimal point. Floating-point variables can be distinguished from integer variables by checking the type specified in their declaration.

The programmer is responsible for ensuring type compatibility of operators and operands in expressions and assignment statements. The compiler can check that programs are type-compatible, and indicate compile time errors when types are not compatible.

The operator + in the expression "X + 1.0" takes floating-point arguments and produces a floating-point value. It has a different meaning from the operator + in the expression "I + 1" which takes integer arguments and produces an integer result. Floating-point addition is in fact implemented by a different machine language operation from integer addition in the machine language of many computers.

Symbols such as + which have different meanings in different contexts are said to be **overloaded**.

The Ada compiler can always determine which meaning of + is intended in any given context by examining the types of its operands.

Note: The ability to determine the types of all operands and to check for type compatibility at compile time is referred to as **strong typing**. Ada is said to be a **strongly-typed language**. Ada allows more rigorous checking of type consistency than in some earlier languages such as Pascal or PL/I.

Which of the following assertions are false?
a) Floating-point literals can be syntactically distinguished from integer literals.
b) Floating-point variables can be syntactically distinguished from integer variables.
c) The operator + in the expression X+2.5*Y has operands of the type FLOAT.
d) A programmer can determine by reading an Ada program whether a given instance of the symbol + represents integer addition or floating-point addition.
e) Type compatibility between the right hand-side of an assignment statement and its left-hand-side can be syntactically determined.

4.2. BOOLEAN Types

The procedure BOOLEANS declares five BOOLEAN variables and assigns values to them.

Example 22: BOOLEAN Variables, Expressions, and Assignment Statements

```
procedure BOOLEANS is
  HUNGRY, SLEEPY, A, B, C: BOOLEAN;   -- declare five BOOLEAN variables
begin
  HUNGRY := TRUE;                     -- HUNGRY becomes TRUE
  SLEEPY := not HUNGRY;               -- SLEEPY becomes FALSE
  A := HUNGRY and SLEEPY;             -- perform logical operations
  B := HUNGRY or SLEEPY;              -- on BOOLEAN variables
  C := HUNGRY xor SLEEPY;             -- and assign values to A, B, C
end;
```

The predefined type BOOLEAN has a two-element value set represented by the literals TRUE, FALSE. Boolean variables may be operated on by the binary logical operators and, or, xor (exclusive or) and the unary operator not. These operators have the standard logical meanings.

A and B is TRUE if both A and B are TRUE and FALSE otherwise.

A or B is FALSE if both A and B are FALSE and TRUE otherwise.

A xor B is TRUE if exactly one of A or B are TRUE and FALSE if both are TRUE or both are FALSE.

not A is FALSE if A is TRUE and TRUE if A is FALSE.

One of the most important uses of BOOLEAN types is in conditional branching statements to decide between alternative courses of action.

Example 23: Use of BOOLEAN Values for Conditional Branching

```
if HUNGRY then     -- if HUNGRY has the value TRUE
  EAT;             -- then perform the action EAT
else
  SLEEP;           -- otherwise perform the action SLEEP
end if;
```

Which of the following assertions are false?

a) The predefined type BOOLEAN has a two-element value set
b) Assignment statements may assign BOOLEAN values to BOOLEAN variables.
c) In the first example the variable A is assigned the value FALSE
d) The variable C is also assigned the value FALSE.
e) Boolean variables may occur in if statements to decide between alternative actions.

4.3. Relational Operators and Relational Expressions

Relational operators such as "<" (less than) take numerical arguments and yield BOOLEAN results. The relational expression "3 < 4" compares its numerical arguments 3 and 4 and yields the BOOLEAN value TRUE because 3 is in fact less than 4.

Ada has six relational operators, each of which takes numerical arguments and yields BOOLEAN values.

```
=  /=  <  <=  >  >=
```

Relational operators may be used to compare the values of two expressions provided the expressions being compared are of the same type.

Example 24: Relational Operators

```
3 < 4                -- arguments are integer literals, value is TRUE
3.5 > 4.5            -- arguments are real literals, value is FALSE
I /= 0               -- TRUE if value of I is not equal to zero
(I < 0) or (I > 9)   -- TRUE except for I in range 0..9
X < Y + 1.0          -- TRUE if value of X is less than value of Y + 1.0
```

Since relational expressions have BOOLEAN values they may be arguments of BOOLEAN operators as in (I < 0) or (I > 9). Relational expressions may also be *assigned as values of BOOLEAN variables.*

```
NEGATIVE := I < 0;   -- NEGATIVE becomes TRUE if and only if I is less than zero
```

Relational expressions, just as other BOOLEAN-valued expressions, can be used to trigger a choice between alternative courses of action.

```
if I < 0 then
  DO_SOMETHING;
else
  DO_SOMETHING_ELSE;
end if;
```

This if statement demonstrates clearly the use of computed numerical results as a basis for execution-time choice among alternative sequences of actions. The ability to make such execution time choices is the basis for "intelligent behavior" by computers.

Which of the following assertions is true?

a) The relational expression 3<4 has numerical arguments and produces a BOOLEAN values.
b) Relational expressions can appear as operands of BOOLEAN operators.
c) Relational expressions can be assigned as values of INTEGER variables.
d) The ability to make execution-time choices based on computed results is a basis for "intelligent behavior" by computers.
e) The relational expressions on the right and left hand side of a relational operator must have the same type.

4.4. Programmer-Defined Types

Programmer-defined types are introduced by type declarations.

```
type SHORT_INTEGER is range -100..100;
```

This type declaration associates the typename SHORT_INTEGER with the type definition "range -100..100" denoting a subrange of the integers. The procedure below illustrates how this type may be used in subsequent object declarations and how objects of the type may in turn be used for computation.

Example 25: Declaration and Use of Programmer-Defined Types

```
procedure TYPES_AND_OBJECTS is              -- a procedure
  type SHORT_INTEGER is range -100..100;   -- with programmer-defined type,
  L,M,N: SHORT_INTEGER;          -- object declaration for the type,
begin                            -- and assignment statements
  L := 3;                        -- that may use integer literals
  M := L + 1;                    -- and integer operations
  N := L + 2 * M;                -- but restrict the value set
end ASSIGN;                      -- to the range -100..100
```

The type declaration for SHORT_INTEGER introduces a new (overloaded) meaning for + that allows it to be used in expressions such as "L + M" for adding two short integers. But "mixed expressions" such as "I + M" with operands of different types are illegal.

The literal 1 in the expression "L + 1" is interpreted as a short integer while the literal 1 in the expression "I + 1" is interpreted as a full integer. Thus literals, just as the operator +, are automatically overloaded when a new type over a subrange of the integers is introduced.

Type declarations in general have the following syntax:

```
type-declaration ::= type TYPE_NAME is type-definition
```

The type definition for SHORT_INTEGER has the form "range -100..100". It specifies that the operations of the defined type are inherited from the integers and that values of variables are restricted to the range -100..100.

Which of the following assertions are false?
a) The literal 53 may be interpreted as a literal of the type INTEGER or as a literal of the type SHORT_INTEGER, depending on context.
b) The operator + may be used to add two integers or two short integers but not an integer and a short integer.
c) The statement N := N + 100; is illegal because it may result in the assignment of an illegal value to N.
d) Both type and object declarations serve to introduce a new identifier and to associate a denotation with the identifier.
e) The type definition range -100..100 defines a type whose values are a subrange of the integers.

4.5. Enumeration Types

Enumeration types have value sets which are defined by explicit enumeration of their elements. The enumeration type DAY has a value set consisting of the seven values MON, TUE, WED, THU, FRI, SAT, SUN.

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);    -- type declaration
```

The procedure WEEKDAYS illustrates the declaration of variables of the type DAY and the assignment of values to these variables.

Example 26: Enumeration Types

```
procedure WEEKDAYS is
  type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);    -- type declaration
  FIRST_DAY, SOME_DAY: DAY;        -- two objects of the type DAY
begin                              -- and a statement sequence
  FIRST_DAY := MON;                -- that assigns values of the type DAY
  SOME_DAY := THU;                 -- to variables of the type DAY
  SOME_DAY := FIRST_DAY;
end;
```

The values of an enumeration type are called **enumeration literals**. Enumeration literals may in general be identifiers or character literals. The values of the type DAY are identifiers. The type VOWEL has character literals as its values.

```
type VOWEL is ('A', 'E', 'I', 'O', 'U');
```

Enumeration types allow variables to have values which are non-numeric. The operations applicable to values of an enumeration type differ from those applicable to numeric variables. For example values of an enumeration type cannot be added or multiplied. But they are more natural than numeric variables in many practical applications. For example variables of the type DAY are useful in computing the work schedules or payroll of an industrial organization.

The value set of an enumeration type is ordered by the order of occurrence of the enumeration values in the type definition. Thus the value set of DAY has the property that MON < TUE and TUE < FRI.

Declaration of an enumeration type introduces not only a new type name but also a set of names of enumeration literals. The literals may be thought of as implicitly declared by their occurrence in the type declaration and have a scope which extends from their point of declaration to the end of the unit in which they are declared. Although they are syntactically indistinguishable from identifiers the compiler can determine from context whether a given identifier is being used as an enumeration literal or as an explicitly declared identifier.

Which of the following assertions are false:
a) Enumeration types are a class of programmer-defined types.
b) Enumeration literals and identifiers have the same syntax.
c) Variables of an enumeration type must take non-numeric values.
d) The value set of the type DAY is ordered with MON < FRI.
e) The enumeration type DAY has seven values in its value set.

4.6. Scalar Types and their Attributes

The class of scalar types comprises the integer types, real types, and enumeration types.

The integer types are the class of all types that have integer values and integer operations, and include the predefined type INTEGER. The real types are the class of all types with real values and associated operations and include the predefined type FLOAT. The enumeration types are the class of all types whose value set is introduced by explicit listing of alternatives, and includes the predefined type BOOLEAN.

All scalar types have the following properties:

1. Scalar types have **ordered** value sets whose elements may be compared by **relational operators**.

2. The value set of any scalar type T has a **least element** denoted by the type attribute T'FIRST and a **greatest element** denoted by the attribute T'LAST.

The first and last elements of the enumeration type DAY can be denoted by the following **type attributes**.

Example 27: The Type Attributes First and Last

```
DAY'FIRST     -- first element of the type DAY (MON)
DAY'LAST      -- last element of the type DAY (SUN)

INTEGER'FIRST    -- first (smallest) integer value, implementation defined
FLOAT'LAST       -- last (largest) floating point value, implementation defined
BOOLEAN'FIRST    -- first Boolean value (FALSE)
```

Type attributes are properties of the type rather than of particular values of the type. They have the form "T'A", where T is the type name, and A is the attribute name. They generally cannot be modified, being accessible in a read-only mode. They allow the programmer to determine properties of the type independently of computations on values of the type.

The class of scalar types is so large that it has only a few type attributes common to all types in the class. We shall see below that the class of discrete types, which is a subclass of the scalar types containing the integer and enumeration types, has a richer set of type attributes.

Which of the following assertions are false?

a) The class of integer types includes the predefined type INTEGER.
b) The class of scalar types includes the predefined type FLOAT.
c) The compiler can always determine whether a given identifier is a type attribute by examining its syntactic form.
d) The relational expression "S < T" has the value TRUE if S and T are of the same scalar type and the value FALSE if S and T are of different scalar types.
e) The value of the attribute INTEGER'FIRST is implementation-defined.

4.7. Discrete Types and their Attributes

The class of discrete types is a subclass of the scalar types that includes integer and enumeration types but excludes real types. Discrete types have ordered value sets with the additional property that every element other than the last has a unique successor.

Every discrete type T has the type attributes T'SUCC and T'PRED which can be applied to values of the type to yield successor and predecessor values.

Example 28: Successor and Predecessor Attributes

```
DAY'SUCC(MON)          -- yields the successor of MON (TUE)
DAY'PRED(TUE)          -- yields the predecessor of TUE (MON)
DAY'SUCC(D)            -- yields the successor of the value of D
DAY'SUCC(DAY'LAST)     -- undefined, last element has no successor
DAY'PRED(DAY'FIRST)    -- undefined, first element has no predecessor
INTEGER'SUCC(I)        -- yields the successor of the integer I
```

The attributes SUCC and PRED are functions which are defined for all discrete types automatically as part of the process of defining the type itself. In this respect they are like the arithmetic operator + which is automatically defined for all numeric types, and the relational operator < which is automatically defined for all scalar types. However, type attributes must be explicitly qualified by a type name while arithmetic and relational operators may be syntactically overloaded so that the type of a particular instance of the operator is implicit in the types of the operands.

Discrete types have other attributes such as POS which determines the ordinal position of a value in the ordered value set, and VAL which determines the value associated with a given ordinal position.

```
DAY'POS(WED)      -- value is 3, WED is third value of type DAY
DAY'VAL(4)        -- value is THU, the fourth value of the type DAY is THU
```

Our purpose is not to give a complete description of attributes of scalar or discrete types but simply to introduce the concept of type classes and type attributes.

Type attributes are useful in programming when it is necessary to determine properties of types independently of computations with objects or values of the type. In addition, a proper understanding of the status of type attributes provides insights into the nature of the notion of type.

Which of the following assertions are false?

a) All discrete types are scalar types.
b) DAY'PRED(MON) is undefined.
c) INTEGER'PRED(INTEGER'LAST) is undefined
d) The type SHORT_INTEGER is an integer type, predefined type, and scalar type.
e) DAY'POS(DAY'VAL(D)) = D for all values D of the type DAY.

Concept Map for Unit 5: Control Structures

The purpose of this section is to introduce control structures that control the order of statement execution. The idea of function declarations with formal parameters and function calls with actual parameters is introduced.

The relation among concepts in this section is as follows:

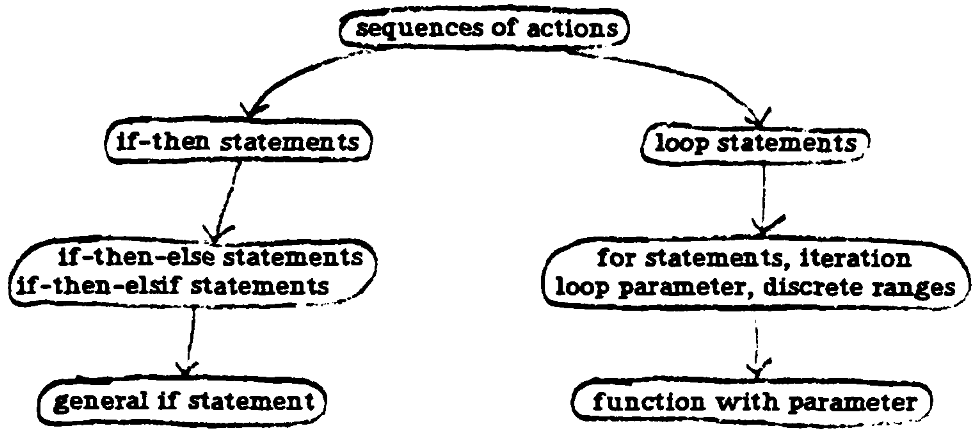


Figure 6: Concept Map for Scalar Types

if statements specify a choice between different courses of action while **loop** statements specify repetitive execution of a sequence of actions.

Several varieties of **if** statements are discussed, including **if-then** statements used to specify optional execution of a component statement, **if-then-else** statements used to specify a choice between two component statements, and **if-then-elsif** statements used to specify a choice among several component statements.

loop statements specify repetitive execution of a sequence of actions. **for** statements are a special kind of **loop** statement which specify repetitive execution of a loop for a sequence of values of a loop parameter. The set of values over which a loop parameter may vary must be a discrete range.

Functions return a value of a specified type and may occur in an expression wherever a value of that type may occur. Functions with formal parameters may be called with different actual parameters on different function calls.

A summation function is defined whose parameter specifies the number of elements to be summed and causes the loop to be executed a different number of times for different actual parameter values.

5.1. Sequences of Actions

Our discussion of types has expanded the class of objects we can describe in the declarative part of a procedure. We shall now turn our attention to the specification of sequences of actions in the statement part.

The assignment statement is the primary mechanism for recording substantive progress in a computation. Long computations generally require the execution of a long sequence of assignment statements each of which records an intermediate result that may be subsequently used.

However the order in which assignment statements are to be executed requires careful specification and generally depends dynamically on initial data values.

The programming language structures concerned with controlling the order of statement execution are called **control structures**. Control structures may be classified into two categories:

conditional branching statements which specify a choice among alternative sequences of action.

loop statements which specify the repetitive execution of sequences of actions

The primary conditional branching statement is the **if** statement. The following **if** statement computes the absolute value of X.

Example 29: One-Branch **if** *Statement (***if-then** *statement)*

```
if X < 0.0 then     -- if X is less than zero
  X := - X;         -- replace X by minus X
end if;             -- otherwise do nothing
```

This **if** statement is an example of an **if-then** statement and has the form:

```
if condition then
  perform action
end if;
```

It allows an execution-time choice to be made between performing an action and omitting it, depending on a computed condition.

Which of the following conditions are false?

a) Assignment statements are the primary mechanism for recording substantive progress in a computation.
b) **if** statements are the primary mechanism for repetitive statement execution.
c) **if** statements and **loop** statements represent two alternative mechanisms for controlling the order of statement execution.
d) Relational expressions may be used to choose among alternative actions of an **if** statement.
e) The variable X in our example is used for both computation and control.

5.2. If Statements

The following two-branch if statement, called an if-then-else statement, determines a choice between two alternative actions. It assigns to Z the maximum of the values of X and Y.

Example 30: Two-Branch if *Statement (*if-then-else *Statement)*

```
if X > Y then       -- if X > Y has the value TRUE
  Z := X;           -- assigns X to Z
else                -- if X > Y has the value FALSE
  Z := Y;           -- assign Y to Z
end if;             -- so that Z is maximum of X and Y
```

Computing the maximum of the three variables A, B,C can be accomplished by the following three-branch if statement.

Example 31: if *Statement with Three Branches*

```
if A > B and A > C then     -- if both A > B and A > C are TRUE
  Z := A;                   -- assign A to Z
elsif B > C then            -- otherwise if B > C
  Z := B;                   -- assign B to Z
else                        -- if none of the above
  Z := C;                   -- assign C to Z
end if;                     -- so that Z is max of A,B,C
```

This if statement illustrates the use of composite conditions such as "A > B" and "A > C" constructed from constituent conditions such as "A > B" by Boolean operators such as and. It illustrates also the keyword elsif which allows multiple branches to be defined, each with a condition which determines whether the statement or sequence of statements associated with that branch is to be executed.

Which of the following assertions are false?

a) An if-then-else statement determines a choice between two alternative sequences of actions.
b) A > B and A > C is true if A is the maximum element of A, B, C.
c) The test B > C is performed only if A is not the maximum element.
d) If A is not the maximum element and B > C then C is the maximum element.
e) Z := C; is executed only if C is the maximum element.

5.3. General Form of the If Statement

The general form of the **if** statement is given by the following syntactic definition.

Example 32: Syntax of the if Statement

```
if-statement ::= if condition then
                   sequence-of-statements
                 {elsif condition then
                   sequence-of-statements}
                 [else
                   sequence-of-statements]
                 end if;
```

An **if** statement always contains a **then** branch. It may have no **elsif** or **else** branches, in which case the choice is between executing and omitting the action in the **then** branch. Each **elsif** branch determines a potential alternative and is executed if the condition in the branch is TRUE and all preceding conditions are FALSE. If an **else** branch is present it occurs last and is executed when all conditions are false. An **else** branch guarantees execution of precisely one branch of the **if** statement, while absence of an **else** branch results in no branch being executed when all conditions are false.

Example 33: if-then-elsif Statement

```
if HUNGRY then
  EAT;
elsif SLEEPY then
  SLEEP;
end if;
```

This results in eating when hungry, sleeping when sleepy but not hungry, and no action (indolence) when neither hungry nor sleepy.

Which of the following assertions are false?

a) The number of conditions in an **if** statement is one greater than the number of **elsif** statements.
b) Conditions may in general have Boolean or numerical values.
c) HUNGRY is a BOOLEAN variable while EAT is a procedure call.
d) The statement "**if** TRUE **then** action **end if**;" has the same effect as unconditional execution of the action.
e) **If** statements without **else** clauses may result in no action being executed.

5.4. The Loop Statement

An if statement causes at most one of its branches to be executed when it is encountered during execution. In contrast, a loop statement specifies multiple executions of a sequence of statements. Loop statements in their simplest form specify unconditional repetitive execution of a sequence of statements.

Example 34: Repetitive Execution with Exit Statement

```
I := 0;                   -- initialize I to zero
loop                      -- and enter a loop
  I := I + 1;             -- which repeatedly increments I by 1
  exit when I = 1000;     -- and exits when I = 1000
end loop                  -- continuing with the next statement
next-statement            -- that textually follows the loop
```

Exit from this loop is determined by an internal condition within the loop (the condition X = 1000).

The for statement is a specialized kind of loop statement which specifies repetitive execution for a sequence of values of a loop parameter as a prefix to the loop.

Example 35: A Simple For Loop

```
SUM := 0;                 -- initialize SUM to zero
for I in 1..5 loop        -- repeat five times with I = 1,2,3,4,5
  SUM := SUM + I;         -- add value of I to sum
end loop;
```

This program first initializes the variable SUM to zero and then executes the statement "SUM := SUM + I;" five times with I taking the values 1,2,3,4,5. On completion, the variable SUM will have the value 1+2+3+4+5 = 15.

I is called the loop parameter and "I in 1..5" is called the loop parameter specification. "1..5" is called a range specification and specifies the range (sequence of values) of the loop parameter.

In the next frame loop parameter specifications whose number of repetitions may depend on the computed value at execution time will be considered.

Which of the following assertions are false?

a) Loops may specify their termination internally by an exit statement or externally as part of the loop parameter specification.
b) A for statement is a special kind of loop statement.
c) Variables that are updated in a loop should be initialized prior to the loop.
d) The variable SUM takes on the five successive values 1,3,6,10,15 during the course of execution of the for loop.
e) The loop parameter specification 1..5 specifies a subrange of the integers.

5.5. The For Statement

The procedure FOR_LOOP sums the first N integers. The summation loop is executed a variable number of times depending on the current value of N.

Example 36: Sum of N Integers

```
procedure FOR_LOOP is       -- procedure containing for loop
  SUM: INTEGER := 0;        -- with initialized declaration of SUM
begin                       -- and a statement part
  for I in 1..N loop        -- with a for statement
    SUM := SUM + I;         -- that is executed N times
  end loop;                 -- to sum the integers 1,2,..,N
ent FOR_LOOP;
```

The three variables N, SUM, and I play different roles in this procedure declaration. N is a non-local variable which must be declared outside the procedure and have a value assigned to it prior to calling the procedure. SUM is a local variable that is declared and initialized in the declarative part of the procedure and repeatedly updated in the statement part of the procedure. I is an implicitly-declared loop parameter that has no explicit declaration.

Non-local, local, and implicitly-declared identifiers represent three alternative mechanisms for introducing identifiers that may be used in a program.

The loop parameter I is considered to be implicitly declared as a local variable of the for statement with a type determined by the element type of the range specification. It takes on the sequence of values of the range specification (1,2,..,N) during the execution of the loop and becomes inaccessible on exit from the loop since its scope is restricted to the for statement. It is a bound variable of the for statement which could be replaced (in its two occurrences) by another variable such as J without changing the effect of the computation.

A loop parameter specification of the form "I in M..N" will cause the sequence of statements in the for loop to be executed N-M+1 times if M <= N. If M > N the loop will be executed zero times.

In the next frame we consider a summation function which inputs the number of items to be summed as a parameter and returns the computed sum as a result to the caller.

Which of the following assertions are false?
a) The non-local variable N must have a value assigned to it prior to calling the procedure.
b) If N has the value 1, then the loop will be executed zero times and the value of SUM will remain zero.
c) Non-local variables are declared externally to a procedure, while local variables are declared internally to a procedure.
d) The loop parameter specification "I in 3..5" would result in three executions of the statement "SUM := SUM + I;", and cause SUM to have the value 12.
e) Replacing the two occurrences of I in the program by J would not change its computational effect.

5.6. Functions with Parameters

Functions are defined by function declarations and called by function calls. They return a value to the caller when their execution terminates.

The function **INTSUM** has a parameter **N** of the type **INTEGER** which specifies the number of elements to be summed. It returns the computed sum as a result to the caller.

Example 37: Sum of First N Integers

```
function INTSUM (N:INTEGER) return INTEGER is
  SUM: INTEGER := 0;            -- local variable initialized to zero
begin
  for I in 1..N loop            -- loop N times
    SUM := SUM + I;             -- accumulate sum
  end loop;
  return SUM;                   -- return value of SUM and exit
end INTSUM;
```

The following assignment statement calls **INTSUM** to sum the first five integers and assigns the result to the variable **SUM1**.

```
SUM1 := INTSUM(5);          -- call with literal parameter
```

Function calls of **INTSUM** can appear in an expression anywhere that an integer variable or integer literal can appear. The following statement computes the sum of the first **K** integers plus twice the sum of the first **K + 3** integers.

```
SUM2 := INTSUM(K) + 2 * INTSUM(K + 3);      -- two calls in one statement
```

The parameter **N** in the function declaration is said to be a **formal parameter** since its only role is to serve as a placeholder for an **actual parameters** transmitted to the function at the time of call. The actual parameters for **INTSUM** can be literals such as 5, identifiers such as **K** or expressions such as **K + 3**.

A function call with a given actual parameter causes the parameter value to be assigned to the formal parameter **N** at the time of function call and used in computing the function value. Execution is terminated by a **return** statement which makes the result available as an operand in an expression.

Which of the following assertions are false?

a) The call **INTSUM(4)** yields the value 10.
b) Formal parameters are place holders for actual parameter values.
c) Actual parameters of functions must be constants.
d) If the statement **SUM := SUM + I;** were replaced by **SUM := SUM + I * I;** the resulting function would compute the sum of the first **N** squares.
e) Exit from a function occurs by executing a return statement.

5.7. Discrete Ranges

The sequence of values which a loop parameter may assume during execution of a **for statement** must be a **discrete range** with the following property.

Definition: A **discrete range** is an ordered sequence of values with a first element, a last element, and the property that all elements other than the last has a unique successor.

Range specifications such as **1..5** and **1..N** determine discrete ranges of integers. However discrete ranges need not be ranges of integers. The ordered sequence of values of an enumeration type determines a discrete range and can be used as a loop parameter specification.

Example 38: Iteration over an Enumeration Type

```
for I in DAY loop
  -- execute statements in loop for each weekday
end loop;
```

This example illustrates that iteration over a discrete range of weekdays is just as meaningful as iteration over a discrete range of integers.

The discrete range associated with the discrete type **DAY** is given by **MON..SUN**. We could have used the discrete range **MON..SUN** in place of the discrete type **DAY** in the above **for** statement. The range of a loop parameter can be specified either by a discrete range or a discrete type.

Every discrete type has a value set **T'FIRST..T'LAST** that is a discrete range, and every discrete range could in principle be named by a discrete type. The discrete range **1..5** could be specified as the following discrete type.

```
type ONE_TO_FIVE is range 1..5;

for I in ONE_TO_FIVE loop ...
```

Discrete types and discrete ranges are important in Ada because they capture the intuitive notion of a sequence of values. The notion of a discrete range arises not only in defining the sequence of values of a loop parameter but also in defining the range of index values of an array.

Which of the following assertions are false?

a) Loop parameter specifications must specify a discrete range over which the loop parameter takes its values.
b) Iteration over discrete ranges of weekdays is just as meaningful as iteration over discrete ranges of integers.
c) **MON..SUN** and **1..5** denote discrete ranges.
d) **type ONE_TO_FIVE is range 1..5;** associates a discrete type with the discrete range **1..5**.
e) A discrete type T has a discrete range **T'FIRST..T'LAST**.

5.8. Syntax of For Statements

A **for** statement consists of the keyword "**for**" followed by a "loop parameter specification" followed by a sequence of statements enclosed by the keywords "**loop**" and "**end loop**".

Example 39: For Statement Syntax

```
for_statement ::= for loop__parameter_spec loop
                    sequence_of_statements
                  end loop;

loop_parameter_spec ::= identifier in [reverse] discrete_range
```

The sequence of statements of the **for loop** is sometimes referred to as its body.

The loop parameter specification consists of an identifier followed by the keyword "**in**" followed by a discrete range optionally preceded by the keyword "**reverse**".

The keyword "**reverse**" causes the loop parameter to take on values of the discrete range in the reverse order. Thus the following program computes the sum "5+4+3+2+1"

Example 40: Reverse For Statement

```
SUM := 0;
for I in reverse 1..5 loop
  SUM := SUM + I;
end loop;
```

I. the present example the keyword **reverse** does not affect the result of the computation, although it does affect the sequence of intermediate values taken by **SUM** during execution of the loop.

One of the most important applications of **for** statements is to iterate over the sequence of components of an array. The declaration of arrays and the use of **for** statements in iterating over sequences of components of an array is discussed in the next instruction unit.

Which of the following assertions are false?

a) The sequence of values taken by **SUM** is **0 5 9 12 14 15**.
b) If **+** were changed to ***** the computed value would be **120**.
c) The reverse of a discrete range is always a discrete range.
d) The keyword **reverse** has no effect on the value computed by a **for** statement.
e) The implicit type of **I** is the element type of the discrete range.

Concept Map for Unit 6: Arrays

The purpose of this unit is to introduce array types, array objects, and the use of for statements to iterate over array objects.

The relation among concepts introduced in this section is as follows:

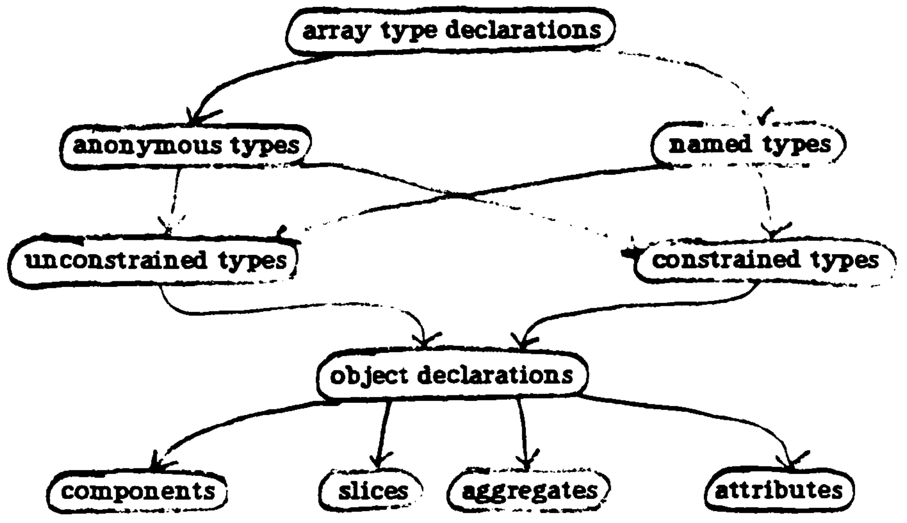


Figure 7: Concept Map for Arrays

We start with anonymous array object declarations and illustrate assignment to array components, iteration over array components, and assignment of array aggregates to array variables.

We then introduce array type declarations and contrast the declaration of anonymous array objects with the declaration of array objects of a named type.

Array attributes are introduced and used to define the range of for statements for iterating over arrays.

Arrays whose index range is an enumeration type are illustrated.

Assignment of array aggregates to array variables and to slices is discussed.

Unconstrained array types whose range is specified at declaration time are introduced. The use of unconstrained array types as parameters of functions is illustrated.

6.1. Introduction to Arrays

An array is a composite object consisting of a *sequence of component objects* all of which have the same type. An array with five components of the type INTEGER may be declared as follows:

```
A: array (1..5) of INTEGER;      -- array object with 5 integer components
```

Components of the array A are denoted by composite names A(I) where I is an index value in the range 1..5. They have the status of variables of the type INTEGER, and may occur on either the right or left-hand side of an assignment statement.

```
A(1) := 17;             -- assign 17 to array component A(1)
A(4) := 2 * A(1);       -- assign 34 to array component A(4)
```

In dealing with arrays we often want to perform similar operations on successive components of an array. The for statement is a control structure specifically designed for this purpose.

Example 41: Iteration over Array Elements

```
for I in 1..5 loop      -- the array specifications 1..5 of this loop
  A(I) := 0;            -- corresponds to the index range
end loop;               -- of the array A
```

This for statement makes use of the index range 1..5 of the array type as its range specification.

In addition to assigning values to array components, Ada also permits direct assignment of values to a complete array.

```
A := (7,6,5,4,3);       -- assignment of array aggregate to array variable
```

The expression (7,6,5,4,3) is called an **array aggregate**. Array aggregates are representations of array values in precisely the same sense that integer literals are representations of integer values. The assignment statement "A := (7,6,5,4,3);" assigns the value represented by the composite literal (7,6,5,4,3) to the composite object A.

Which of the following assertions are false?

a) "array (1..5) of INTEGER;" has an index range "1..5"
b) The components of A are A(1), A(2), A(3), A(4), A(5).
c) Array aggregates may be assigned as values to array literals
d) A(2) is a variable of the type INTEGER.
e) For statements are designed to facilitate iteration over array data structures.

8.2. Type and Object Declarations for Arrays

The declaration "A: array (1..5) of INTEGER;" associates the array object A directly with a type definition. It is called an anonymous array declaration because it does not use a type name in declaring the array object A.

Array objects introduced by anonymous declarations are always of different type.

Example 42: Anonymous Object Declarations

```
A: array (1..5) of INTEGER;          -- A and B are of different anonymous types
B: array (1..5) of INTEGER;
C, D: array (1..5) of INTEGER;        -- even C and D have different types
```

Instead of declaring an array object directly we can declare an array type from which array objects may be subsequently created by object declarations.

```
type ARRAY_TYPE is array (1..5) of INTEGER;
```

This type declaration serves to associate a name with the type definition "array (1..5) of INTEGER;". Array objects having the index range and component type of this type definition can be defined as follows:

Example 43: Named Array Type Declarations

```
A: ARRAY_TYPE;        -- A and B are of the same type
B: ARRAY_TYPE;
C,D: ARRAY_TYPE;      -- C and D are of the same type as A and B
```

Array objects of anonymous type avoid the overhead of introducing an explicit type name and are useful in applications involving computation on a single array in a single program module. But computation that requires several arrays of the same type or that requires the transmission of array parameters between program modules requires array objects to have a named type.

Which of the following assertions are false?

a) Arrays introduced by different anonymous declarations always have a different type.
b) Array type declarations associate a name with a type definition
c) "A: array (1..5) of INTEGER;" declares an array object of anonymous type.
d) Having the same index range and component type is a necessary but not sufficient conditions for two array objects to have the same type.
e) Arrays passed as parameters between program modules cannot have an anonymous type definition.

6.3. Array Attributes

In performing computations on an array it is sometimes useful to refer to properties of the array data structure. Such properties are called **array attributes.** Three useful attributes of the index range of an array object are its first value, last value, and range.

```
A'FIRST    -- first value in index range of the array A
A'LAST     -- last value in index range of the array A
A'RANGE    -- index range of the array A
```

Array attributes are denoted by composite names consisting of an array name followed by an apostrophe followed by an attribute name. They have the same syntax as type attributes but are associated with array objects rather than with types. The values of array attributes come into existence when the object declaration for the array is executed and cannot be modified by assignment. They can be used to determine properties of the array such as the number of its components.

```
SIZE_OF_A := A'LAST - A'FIRST + 1;      -- number of components of A
```

The range attribute of an array can be used as the range specification of a for statement. The example below assumes that the components of A have previously assigned values.

Example 44: Using the Range Attribute

```
SUM := 0;
SUMSQ := 0;
for I in A'RANGE loop
  SUM := SUM + A(I);
  SUMSQ := SUMSQ + A(I) * A(I);
end loop
```

The use of A'RANGE allows the programmer to specify iteration over the sequence of components of an array without explicit knowledge of the index range or number of elements.

Which of the following assertions are false?

a) The number of index values in A'RANGE is A'LAST-A'FIRST.
b) A'RANGE is a read-only attribute.
c) Iteration over A'RANGE allows the programmer to sum all elements of an array without actually knowing how many there are.
d) The components of A must be assigned values before their sum can be computed.
e) The value of SUM will be 25 after execution of the for loop.

6.4. Index Ranges of Enumeration Types

The index range of an array need not be a range of integers. It may be any discrete range, including the value set of an enumeration type.

The index range of the array type HOURS_WORKED is the days of the week.

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);    -- an enumeration type type
HOURS_WORKED is array (DAY) of FLOAT;      -- an array type
```

This array type may be used as a basis for defining array objects that record the number of hours worked by employees in a company.

```
SMITH, JONES: HOURS_WORKED;     -- array objects with index range DAY
```

The components SMITH and JONES can be initialized by assignment:

```
SMITH(MON) := 7.5;  -- assign value to MON component of SMITH
JONES(SAT) := 4.3;  -- assign value to SAT component of JONES
```

The total number of hours worked by SMITH and JONES can be computed by a for loop which iterates over the index range DAY.

Example 45: Summation over an Enumeration Range

```
SUM_SMITH := 0.0;
SUM_JONES := 0.0;
for D in DAY loop
  SUM_SMITH := SUM_SMITH + SMITH(D);
  SUM_JONES := SUM_JONES + JONES(D);
end loop;
```

This for statement uses the index range of the array type of SMITH and JONES as its range specification. Since SMITH and JONES both have the same index range DAY we can perform summation for both arrays in the same for loop.

Which of the following assertions are false?

a) The array object SMITH has the range MON..SUN.
b) SMITH(MON) is a variable of the type FLOAT.
c) SMITH is an array with seven components.
d) SMITH and JONES have the same index range and element type but may have different values.
e) The index variable D ranges over a sequence of integers.

6.5. Aggregates, Slices, and Complete Arrays

Ada has a variety of facilities for operating directly on arrays and subarrays which encourage the programmer to think of arrays as single data objects rather than as collections of components.

Literals which represent values of complete arrays are called **array aggregates**. Array aggregates may be assigned to array variables of compatible type in much the same way that integer literals are assigned to integer variables.

```
A := (7,6,5,4,3);    -- assign array aggregate to array variable
```

Array variables may appear as operands on both the right and left hand side of an assignment statement.

```
B := -A;         -- B is assigned the value (-7,-6,-5,-4,-3)
B := 2 * A;      -- B is assigned the value (14,12,10,8,6)
```

Ada allows the meaning of - and * to be extended to arrays. Unary minus causes negation of all the elements of the array, while multiplication by a scalar causes all elements to be multiplied by the scalar. The extension of operators to operands of a new type is called overloading.

Ada allows us to treat subarrays of contiguous components of an array as composite variables having the same status as complete arrays.

```
A(2..4)    -- subarray consisting of A(2), A(3), A(4)
```

The subarray A(2..4) is called a **slice** of the array A. Slices may appear as operands on both the left and right hand sides of an assignment statement.

```
A(1..2) := (35, 17);  -- assign the aggregate (35, 17) to the slice A(1..2)
A(4..5) := 2 * A(1..2);    -- assign the aggregate (70,34) to the slice A(4..5)
```

Which of the following assertions are false?

a) The composite literal (1,2,3,4,5) may be assigned as a value to the composite array object A.
b) Arrays may appear as operands of arithmetic operators.
c) Arithmetic operators may be overloaded to operate on arrays and slices.
d) A slice is a contiguous subsequence of components of a one dimensional array.
e) The assignment A(2..4) := A(4..5); is legal because all slices of an array have the same type.

6.6. Unconstrained Array Types

Ada allows the index range of an array type to remain unspecified in an array type declaration so that it can be defined separately for each object of the type.

```
type VECTOR is array (INTEGER range <>) of INTEGER;
```

The type VECTOR is called an **unconstrained array type**. Its index range specification "INTEGER range <>" indicates that the index range of objects of the type will be a subrange of the integers, but allows the actual subrange to be specified at object declaration time rather than at the time the type is declared.

```
V: VECTOR(1..5);     -- V has index range 1..5, five components
W: VECTOR(1..10);    -- W has index range 1..10, ten components
```

The use of unconstrained array types may be illustrated by the function VECSUM which has a parameter of the type VECTOR and returns the INTEGER sum of the components of the vector as a result.

```
function VECSUM( X: VECTOR) return INTEGER;
```

Since VECTOR is unconstrained VECSUM can be called with vectors of different sizes.

```
X := VECSUM(V)       -- sum the components of the five-element vector V
Y := VECSUM(W)       -- sum the components of the ten-element vector W
```

This function would be much less useful if the type VECTOR were constrained to include only vectors of a given size. In this case it would be necessary to define separate functions for ten component and eleven component vectors.

There are many other applications, such as sorting and linear algebra, where the ability to write subprograms that operate on arrays of different size is essential. If we adopt the Ada principle that all parameters are of a fixed type the concept of type must be sufficiently broad to allow a single type to include arrays of different sizes.

Pascal suffers from the fact that its concept of an array type is too inflexible. In Pascal two arrays with different index ranges cannot have the same type and subprograms that sum or sort arrays of arbitrary size are accordingly difficult to define. The Ada concept of unconstrained array types was developed in part as a response to this deficiency of Pascal.

Note: The unconstrained array type VECTOR may be thought of as a **parameterized type**. The specification "INTEGER range <>" plays the role of a formal parameter in the type declaration. It is bound to actual parameters such as 1..5 or 1..10 at object declaration time.

Which of the following assertions are false?
a) The type VECTOR has an unspecified index range.
b) Objects of the type VECTOR may be arrays of different sizes.
c) Functions with an unconstrained array parameter may be called with actual parameter arrays of different sizes.
d) "K := VECSUM(V) + VECSUM(W);" assigns to K the sum of the sums of V and W.
e) In Pascal vectors with different index ranges have different types.

Concept Map for Unit 7: Subprograms and Parameters Passing

The purpose of this unit is to consolidate concepts relating to subprograms and parameter passing, as indicated in the following concept map.

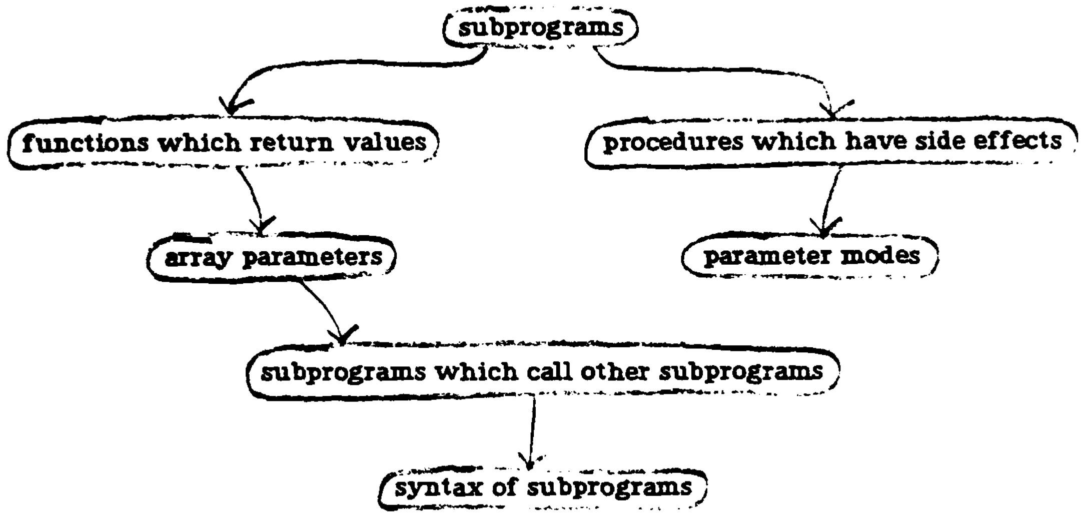

subprograms
functions which return values
procedures which have side effects
array parameters
parameter modes
subprograms which call other subprograms
syntax of subprograms

Figure 8: Concept Map for Subprograms and Parameter Passing

We start with a function for summing components of an array. It has an unconstrained array as a parameter so that actual parameters can be arrays of different sizes on different instances of call.

An alternative implementation by a procedure is presented which requires an extra parameter for storing the the value of the computed sum. We demonstrate that summation is more naturally implemented by a function than by a procedure.

The parameter modes **in**, **out** and **in out** are introduced. The **in out** mode is illustrated by a swap procedure. Swapping is more naturally implemented by a procedure than by a function.

A MAX_INDEX function is defined for finding the maximum of a set of vector components. A SORT procedure is then defined which calls the previously defined SWAP and MAX_INDEX subprograms in sorting the components of its array parameter.

The SORT subprogram is our first example of a relatively non-trivial computation. It brings together almost all the language features we have learned so far and illustrates them in the context of a single subprogram. The SORT subprogram illustrates expressions, statements, **if** and **for** statements, functions, subprograms, array parameters, and slices.

Sorting is a good example both because it brings what we have learned so far into focus and because it is a very widely used practical application. The reader who has persevered to this point has mastered most of the basic language ideas of predecessors of Ada and is ready to start learning about more advanced language features.

7.1. Function for Summing the Elements of an Array

The function VECSUM below has a parameter V of the unconstrained array type VECTOR and may be called to sum vectors of different sizes.

Example 46: Function for Summing Elements of a Vector

```
function VECSUM(V:VECTOR) return INTEGER is     -- VECTOR parameter
 S : INTEGER := 0;                   -- local variable S, initialize to 0
 begin                               -- the statement sequence
  for I in V'RANGE loop              -- loops over actual parameter
   S := S + V(I);                    -- adding values to S
  end loop;
  return S;                          -- and returns with value of S
end SUM;
```

The index range V'RANGE of the for loop is determined by the actual parameter at the time of function call.

A call of VECSUM for an actual parameter A has the following form:

```
SUM1 :=  VECSUM(A);          -- call of VECSUM with actual array parameter
```

This call of VECSUM with the actual array parameter A causes both the values of the array and its index attributes to become accessible. VECSUM will compute the sum of components A(I) in the range A'RANGE. The number of times the for loop is executed will depend on the number of components of A.

The following statement contains two calls of VECSUM for summing two different vectors A and B which may in general have different numbers of components.

```
SUM2 :=  VECSUM(A) + 2 * VECSUM(B);      -- two calls of VECSUM
```

Calls of VECSUM return a value of the type INTEGER and may be used in an expression wherever an integer variable or integer literal may occur.

Which of the following assertions are false?

a) Calls of VECSUM return a value of the type INTEGER.
b) V is a formal parameter whose actual parameter values may be arrays of different sizes on different instances of call.
c) V'RANGE is a formal range whose actual range is determined by the index range of the actual parameter.
d) The type of S must be the same as the type of VECTOR.
e) If S := S + V(I); were changed to S := S + V(I) * V(I); the function VECSUM would compute the sums of squares of its actual parameter.

7.2. A Procedure for Summing Vectors

Let's see what would happen if we used a procedure rather than a function to sum the components of a vector.

Procedure calls do not return values on return from a procedure call. However they may have output parameters that allow values computed during execution of the procedure to be stored externally and used after execution of the procedure is completed. The procedure **PROCSUM** has an output parameter **S** (of the mode out) in which the computed sum may be stored prior to return from the procedure.

Example 47: Procedure which Computes Vector Sum

```
procedure PROCSUM (V:VECSUM; S: out INTEGER) is     -- extra output parameter
  TEMP: INTEGER := 0;             -- TEMP is used to accumulate the sum
begin
  for I in V'RANGE loop
    TEMP := TEMP + A(I);
  end loop;
  S := TEMP;               -- assign computed sum to the output parameter
end PROCSUM;
```

Procedures are terminated by executing a **return** statement or when control reaches their **end** statement. This occurs after executing "**S := TEMP;**" in the present procedure.

The procedure **PROCSUM** may be called as follows:

```
PROCSUM(A, SUM);  -- store computed sum of A in variable SUM
```

The sum of components of a vector is more naturally specified as a function than a procedure because the sum may naturally be thought of as a value. However for subprograms whose computational effect is not that of computing a value, procedures may be a more natural form of specification.

PROCSUM modifies its environment by storing the computed sum in its output parameter. Such modification of the environment is called a **side effect**. Functions do not need to have side effects because they can achieve their effect by returning a value that may be used in an expression for further computation. Procedures cannot return a value and can affect subsequent computations only by means of side effects.

Which of the following assertions are false?

a) Procedures can perform any computation performed by functions.
b) Functions are more natural than procedures for performing certain computations.
c) The call **PROCSUM(A, SUM);** has a side effect which modifies the variable **SUM**.
d) The call **PROCSUM(A, SUM);** has a side effect which modifies the array **A**.
e) If **A** has zero components then **PROCSUM(A, SUM);** assigns zero to the variable **SUM**.

7.3. Parameter Modes

Formal parameters of subprograms have both a type which specifies its applicable operations, and a mode which specifies the manner in which values are communicated between the calling and the called program.

There are three parameter modes, **in**, **out**, and **in out**.

Example 48: Parameter Modes

in	The formal parameter is a constant and permits only reading of the associated actual parameter
out	The formal parameter is a variable and permits updating of the value of the associated actual parameter. If the actual parameter is not updated during a given call its value is undefined.
in out	The formal parameter is a variable and permits both reading and updating of the associated actual parameter.

The procedure SWAP below has parameters of the mode "in out" because swapping of two variables requires both reading and updating of the variables.

Example 49: Swapping of Two Variables

```
procedure SWAP(X,Y: in out INTEGER) is -- two parameters of mode in out
  LOCAL: INTEGER;                -- with a local variable
begin                            -- and a sequence of statements
  LOCAL := X;                    -- which uses LOCAL for temporary storage
  X := Y;
  Y := LOCAL;                    -- in interchanging the values of X, Y
end SWAP;
```

Note that the swap procedure modifies its environment but does not compute any result that is naturally thought of as a value. It is therefore more naturally defined as a procedure than as a function.

Which of the following assertions are false?

a) Parameters of the mode **in** cannot be updated.
b) Parameters of the mode **out** must be variables.
c) Parameters of the mode **in out** can be used for both input of data on entry to the subprogram and output of results on exit from the subprogram.
d) The call SWAP(A(I), A(J); interchanges the values of the Ith and Jth components of A.
e) The call SWAP(3, 5); interchanges the values 3 and 5.

7.4. Computing the Maximum

Finding the maximum of the components of a vector requires a loop with a more complex structure than that required to sum the components of the vector.

Example 50: Finding the Maximum of a Vector

```
MAX := V(1);                  --initialize MAX to V(1)
for I in 2..10 loop           --and compare successive components against MAX
  if V(I) > MAX then          --if current V(I) greater than MAX so far
    MAX := V(I);              --assign new largest value to MAX
  end if;                     --otherwise leave old maximum value
end loop;                     --exit when discrete range exhausted
```

MAX is used to keep track of the maximum value found so far. It is initially set to V(1) and is replaced whenever a component greater than its current value is found. This ensures that at any intermediate point of the computation MAX is the maximum of the components examined so far, and that it is the maximum of all components at the end of the computation.

The value of MAX provides us with a knowledge of the maximum value but not with the position of the maximum component in the vector. In order to determine the position we may compute the index of the maximum value.

Example 51: Index of Minimum Vector Element

```
INDEX := V'FIRST;             --initialize INDEX to first index value
for I in V'RANGE loop         --V(INDEX) will hold maximum vale so far
  if V(I) > V(INDEX) then     --if current V(I) > max so far
    INDEX := I;               --set INDEX to new largest value
  end if;                     --otherwise leave old max index value
end loop;                     --exit when all index values are examined
```

INDEX is initially set to the index of the first component and is updated whenever the new component value is greater than all previous values. On completion, INDEX will contain the index of the maximum element.

Which of the following assertions are false?

a) MAX is initialized to the value of the first component of V.
b) MAX is updated whenever a new value is greater than all previous values.
c) Both programs compute the value of the maximum of the vector V.
d) Both programs contain an if-then statement nested in a for statement.
e) For a given vector V the number of times "MAX := V(I);" is executed in the first program is equal to the number of times "INDEX := I;" is executed in the second program.

7.5. Maximum Index for Arrays and Slices

A function for computing the index of the maximum element of a vector can be defined as follows.

Example 52: MAX_INDEX *Function*

```
function MAX_INDEX(V:VECTOR) return INTEGER is   --function specification
  INDEX: INTEGER := V'FIRST          --local variable INDEX initialized to V'FIRST
begin                                --beginning of statement sequence
  for I in V'RANGE loop              --for I in range of vector parameter
    if V(I) > V(INDEX) then          --if current V(I) > maximum so far
      INDEX := I;                    --assign new maximum index
    end if;                          --otherwise leave old maximum index
  end loop;                          --exit loop when all index values examined
  return INDEX;                      --return with value of maximum index
end MAX_INDEX;                       --end of function MAX_INDEX
```

The usefulness of returning the index of the maximum element of a vector rather than the maximum value itself is illustrated by the following program fragment for interchanging the values of the first element and maximum element of a vector.

Example 53: Interchange Maximum Element with First Element

```
K := MAX_INDEX(A);       --assign maximum index of A to K
TEMP := A(K);                  --store maximum value in TEMP
A(K) := A(1);            --place A(1) in old maximum position
A(1) := TEMP;                  --store maximum value as A(1)
```

Calls of MAX_INDEX with parameters that are slices of an array A are illustrated below.

```
MAX_INDEX(A(I..J))          -- compute MAX_INDEX of the slice A(I..J)
MAX_INDEX(A(I..A'LAST))     -- A(I..A'LAST) is a "terminal" slice of A
```

In the next frame we shall discuss an algorithm for sorting by finding the maximum of a sequence of slices of the vector being sorted.

Which of the following assertions are false?

a) The formal parameter of MAX_INDEX is an unconstrained array.
b) The number of times the for loop is executed depends on the component type of the actual parameter.
c) The statement "A(MAX_INDEX(A) := 0;" replaces the maximum component of A by zero.
d) The actual parameter of MAX_INDEX can be a slice of an array of the type VECTOR.
e) "A(I..A'LAST)" is a slice whose last component is the last component of A.

7.6. Sorting by Successive Maxima

Let's illustrate sorting by successive maxima by showing how the five component vector "3 1 7 6 5" is sorted into the vector "7 6 5 3 1".

We first find the index "3" of the maximum "7" of "3 1 7 6 5" using the MAX_INDEX function. The first and third components are interchanged, yielding the vector 7 1 3 6 5.

This ensures that the maximum component is in the first position and we have reduced the problem to finding the maximum of the slice A(2..5) consisting of the four components 1 3 6 5. The maximum index of this slice is 4. Interchanging the second and fourth elements yields 7 6 3 1 5.

Finding the maximum index of A(3..5) and interchanging the maximum and third element yields 7 6 5 1 3.

Finally, finding the maximum index of A(4..5) and interchanging the fourth and fifth elements yields the completely sorted sequence 7 6 5 3 1.

A sorting procedure which realizes this algorithm for vectors with integer elements is given below. The MAX_INDEX function is used to find the maximum of successive slices A(I..A'LAST), and the procedure SWAP is used to interchange the maximum element of each slice with the first element of the slice.

Example 54: Sort Procedure

```
procedure SORT(A:in out VECTOR) is      --specification of SORT procedure
  K: INTEGER;                           --local variables K, TEMP
begin                                   --beginning of statement sequence
  for I in A'RANGE loop                 --for I in range of vector A
    K := MAX_INDEX(A(I..A'LAST));       --find max index of slice
    SWAP(A(I), A(K));                   --swap with first component
  end loop;                             --exit from loop
end SORT;                               --end SORT procedure
```

Our sorting algorithm has an inner loop which swaps the Ith component with the maximum of the slice A(I..A'LAST) for successive values of I in A'RANGE.

Note: The two statements in the inner loop of this program could be replaced by the following single statement.

```
SWAP(A(I), A(MAX_INDEX(A(I..A'LAST)));
```

This would eliminate the need for a temporary variable K, and would result in a shorter program. But the resulting program would be more difficult to understand than the programs in our example.

Which of the following assertions are false?

a) Sorting involves both reading and updating of the vector being sorted.
b) The number of components in A(I..A'LAST) increases as I increases.
c) SWAP(A(I), A(K)); swaps the first and maximum components of A(I..A'LAST).
d) SORT is defined in terms of the subprograms MAX_INDEX and SWAP.
e) SORT may be used to sort vectors of different size and type.

7.7. Syntax of Functions and Procedures

Both procedures and functions are subprograms consisting of a subprogram specification followed by a declarative part followed by a sequence of statements enclosed in begin - end parentheses.

Example 55: Structure of Subprograms

```
subprogram ::= subprogram_specification is
                 declarative_part
               begin
                 sequence_of_statements
               end [subprogram_identifier];
```

Procedures and functions have a similar structure in their declaration part and statement part but differ in their subprogram specification.

```
subprogram_specification ::=
   procedure identifier [formal_part] |
   function designator [formal_part] return type_mark
```

Syntactic differences between functions and procedures include the fact that functions require the keyword **return** and a type mark such as INTEGER following the parameter specification. Function names are richer than procedure names since designators include operator symbols as well as identifiers.

```
designator ::= identifier | operator_symbol
```

The formal parameter part of procedures and functions is similar, consisting of a sequence of formal parameter specifications separated by semicolons and enclosed in parentheses.

```
formal_part ::=  (parameter_spec {; parameter_spec})
```

A parameter specification is an identifier list with associated mode and type that may optionally be initialized to an expression.

```
parameter_spec ::=  identifier_list : [mode] type_mark [:= expression]
type_mark ::= type_name | subtype_name
```

The above syntax definitions have a few undefined terms such as "expression", "subtype name", etc. These have been left undefined because portions of the language required to make sense of these definitions have not yet been completely covered.

Which of the following assertions are false?
a) Procedure and function declarations have the same syntax.
b) INTEGER is a type mark.
c) + may be a function name but not a procedure name.
d) I,J: INTEGER := 0 is a valid parameter spec.
e) Functions need not have parameters but must return a result.

Answer Key

Note on the Role of Assertions

The assertions at the end of each frame are a mechanism for teaching rather than testing. They provide an opportunity to make additional true statements about the material in each frame. The search for false assertions provides an incentive for careful reading that enhances the learning process. Generally there is just one false assertion. But occasionally there is none or more than one just to keep the reader honest.

Assertions serve to consolidate and review previously presented material. They are intended to be easy rather than mind-stretching, and to build up the confidence of the reader in using Ada terminology. At this early stage of learning Ada, practice in using Ada terminology is as important as writing Ada programs.

Reviewing the answers is an important part of the learning process. Looking up the answer confirms not only the falsity of false assertions but also the truth of true assertions. Answers occasionally discuss issues that go beyond the material presented in the frames. Such discussions could be augmented in subsequent drafts of the material as experience is gained concerning points readers wish to see discussed.

Answers for Unit 1: Lexical Elements

1.0. Warming-up Exercise
d) Rules of semantics determine the "meaning" of programs.
Discussion: Meaning is an elusive concept which cannot be precisely defined. As a first approximation we can think of the meaning of a program construct as its computational effect. Thus the meaning of the expression "X+Y" is the relation between values of X,Y and the value of "X+Y". The meaning of a function such as SQRT(X) is similarly determined by the relation between values of X and values of the result that is computed for each value of X.

1.1. The Ada Character Set
d) The character string "3.1416" contains six characters. Five are digits and one is a special character.

1.2. Lexical Elements and Separators
c) Extra spaces do not change the effect of a program. However in cases of ambiguity at least one space is necessary. Thus in "**if** X = Y **then** ..." at least one space is needed between **if** and X and between Y and **then**.

1.3. Identifiers
d) Predefined identifiers such as INTEGER can be redefined by the user. Only reserved identifiers cannot be redefined.
Discussion: It is inadvisable to redefine predefined identifiers without good reason. Thus the predefined type INTEGER should never be used as the name of a variable. But INTEGER could be redefined if there are strong reasons for integers to have a precision other than the standard implementation-defined precision.

For function identifiers such as SQRT new meanings can be added without invalidating previous meanings. Thus if SQRT is defined in the library only for floating-point numbers of standard precision then a new definition of SQRT for numbers of double precision does not invalidate the previous meaning. Adding new meanings of an identifier without invalidating previous meanings is referred to as **overloading**.

1.4. Numeric Literals
a) 1/10000 is greater than 1/100000.

1.5. Character and String Literals
d) "I think therefore I am" has 22 characters. 18 are letters and 4 are special characters.

1.6. Comments
c) Comments are useful during program maintenance. Their removal does not increase program efficiency.

1.7. Positional Number Representation
b) The value of a digit in positional number representation depends on its base. If the base is n, then each digit position to the left is worth a factor of n more than its right neighbor.

1.8. Based Literals
c) 15#EF# is illegal since F has no meaning in base 15. The hexadecimal digit F stands for 15 base 16, but has no meaning in numbers with a lower base. Numbers in base N can contain only the digits 0,1,...,N-1.

Answers for Unit 2: Syntactic Notation

2.1. Productions which name sets of characters.
e) If the sets associated with X and Y overlap, then X|Y will have fewer elements than the sum of X and Y. Thus if X ::= A|B and Y ::= B|C then X|Y is the three-element set {A,B,C}.

Note that the number of elements in X|Y is always less than or equal to the sum of the number of elements in X and Y.

2.2. The Concatenation Operator
c) If X has M elements and Y has N elements then X Y has M * N elements. Thus, if one of the two sets associated with X and Y is empty, then the set X Y will be empty. If X has a single element and Y has at least one element then the number of elements in X Y will be equal to the number of elements in X. The number of elements in X Y is greater than the number of elements in X and Y only if both X and Y have more than one element.

2.3. The Syntax of Integers and Identifiers
d) The production abc_string ::= {a|b|c} asserts that abc_string consists of an arbitrary number of as, bs, and cs.

2.4. The Syntax of Numeric Literals
This frame has two false assertions:
d) Neither 2#FF# nor 3#FF# are valid based literals.
e) The based literal 3#0.1#, which has the value one third, cannot be represented by a finite decimal literal.

Answers for Unit 3: Expressions, Statements, and Declarations

3.1. Integer-Valued Expressions
c) (4*3)**2 = 12**2 = 144, while 4*(3**2) = 4*9 = 36.

3.2. Expressions and Variables
e) When I = 3 and J = 4, I**J = 3**4 = 81 and J**I = 4**3 = 64.

3.3. Assignment Statements and Memory Cells.
none

3.4. Object Declarations
b) The declaration of a variable creates an object and associates it with a declared identifier. But declarations such as I: INTEGER; leave the variable I uninitialized.

Ada permits variables to be initialized at their point of declaration by declarations such as I: INTEGER := 0;.

3.5. Blocks with Local Variables
d) Variables in a block may include both **local variables** declared in its declaration part, and **non-local variables** declared in outer layers of the program.

3.6. Parameterless Procedures
c) I,J,K are **local variables** - not parameters. ASSIGN is a **parameterless procedure.** The relation between parameters and local variables will be discussed later. One of the principal differences is that parameters are part of the user interface, while local variables are known only within the procedure.

3.7. Review of Declarations and Statements
b) Computational entities declared locally within a procedure come into existence when the procedure is called during execution, and disappear when execution of the procedure is completed. Each call of the procedure has its own copy of local entities.

Answers for Unit 4: Introduction to Types

4.1. Floating-Point Variables
b) Floating-point and integer variables cannot be inherently distinguished by their syntactic form. However, for any given occurrence of a variable in the program text it is possible to determine its associated declaration at compile time, and thereby to determine its type.

4.2. Boolean Types
d) Since HUNGRY = TRUE and SLEEPY = FALSE, the variable C is assigned the value TRUE.

4.3. Relational Operators and Relational Expressions
c) Relational expressions can be assigned as values of Boolean variables, but not as values of INTEGER variables.

4.4. Programmer-Defined Types
c) The statement N := N + 100; is not syntactically illegal. We cannot determine at compile-time whether the result of executing N := N + 100; satisfies the constraints for SHORT_INTEGER. A run time check is needed to compute if this statement assigns an acceptable value to N.

4.5. Enumeration Types
b) Enumeration literals may be identifiers or character literals. Enumeration literals such as **MON** are syntactically indistinguishable from identifiers. Thus statements such as X := MON; will cause assignment of the literal **MON** if X is an enumeration type, and assignment of the value of the variable **MON** if X is a variable.

4.6. Scalar Types and their Attributes
d) The value of the relational expression S < T depends on the values of the variables S and T. The value is undefined when S and T are of different types.

4.7. Discrete Types and their Attributes
c) INTEGER'LAST has a predecessor which is implementation-defined. Only INTEGER'FIRST has no predecessor.

Answers for Unit 5: Control Structures

5.1. Sequences of Actions
b) **if** statements are mechanisms for choosing among alternative actions. **loop** statements are the mechanism for repetitive statement execution.

5.2. If Statements
d) If A is not the maximum and B > C then B is the maximum element.

5.3. General Form of the **if** Statement
b) The value of a condition must be BOOLEAN (TRUE or FALSE).

5.4. The **loop** Statement
d) The variable SUM takes on the six values 0,1,3,6,10,15.

5.5. The **for** Statement
b) If N has the value 1 the loop will be executed once and cause SUM to have the value 1.

5.6. Functions with Parameters
c) Actual parameters of functions may be variables of any type. However, assignment to actual parameters is generally prohibited so that the values of variables generally remain constant during execution of the function.

5.7. Discrete Ranges
none

5.8. Syntax of **for** Statements
b) The product 0*1*2*3*4*5 is zero, not 120.
d) The keyword **reverse** may in general affect the computational result of a **for** loop. It makes no difference in a summation loop or in any computation where the end result does not depend on the order in which statements of the loop are executed. It makes a difference in the following cases:

(1) Replace SUM := SUM + I; by SUM := SUM ** I; (2**3**4**5 is approximately 10**18, while 5**4**3**2 is approximately 10**11).

(2) Consider loop with a single statement A(I) := A(I+1); (ascending I has the expected effect of shifting to the right, while descending I will cause all components to have the same value).

Answers for Unit 6: Arrays

6.1. Introduction to Arrays
c) Array aggregates may be assigned as values of array variables.

6.2. Type and Object Declarations for Arrays.
None

6.3. Array Attributes
a) The number of index values in A'RANGE is A'LAST - A'FIRST + 1.
e) The value of SUM will be 1+4+9+16+25 = 55.

6.4. Index Range of Enumeration Types
e) The index value D ranges over a space of enumeration literals.

6.5. Aggregates, Slices, and Complete Arrays.
e) Only slices with equal numbers of elements can be assigned to each other.

6.6. Unconstrained Array Types
None

Answers for Unit 7: Subprograms with Array Parameters

7.1. Function for Summing Elements of an Array
d) The type of S must be the same as the type of the components of VECTOR.

7.2. A Procedure for Summing Vectors
d) The call PROCSUM(A, SUM) modifies the value of SUM but not the value of A.

7.3. Parameter Modes
e) The call SWAP(3, 5) is not valid because actual in out parameters must be variables.

7.4. Computing the Maximum
c) Only the the first program computes the value of the maximum component. The second program computes the index of the maximum component.

7.5. Maximum Index for Arrays and Slices
b) The number of times the for loop is executed depends on the index range of the actual parameter.

7.6. Sorting by Successive Maxima
b) The number of components of A(I..A'LAST) decreases as I increases.

7.7. Syntax of Functions and Procedures
a) Function and procedure specifications differ in their syntax in several respects. The keywords **function** and **procedure** are different. Function names may be identifiers or operator symbols, while procedure names are restricted to identifiers. Function specifications must have the keyword return followed by the type of the object returned by the function, while procedures do not return an object.

END

FILMED

1-85

DTIC